İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

SİMÜLASYON YAZILIMLARINDA KOD KLONLARI

YÜKSEK LİSANS TEZİ

Merve ASTEKİN

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

AĞUSTOS 2012

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

SİMÜLASYON YAZILIMLARINDA KOD KLONLARI

YÜKSEK LİSANS TEZİ

Merve ASTEKİN
(504101511)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Prof. Dr. Muhittin GÖKMEN

AĞUSTOS 2012

İTÜ, Fen Bilimleri Enstitüsü'nün 504101511 numaralı Yüksek Lisans Öğrencisi **Merve ASTEKİN**, ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı **"SİMÜLASYON YAZILIMLARINDA KOD KLONLARI"** başlıklı tezini aşağıda imzaları olan jüri önünde başarı ile sunmuştur.

Tez Danışmanı :	**Prof. Dr. Muhittin GÖKMEN** İstanbul Teknik Üniversitesi	.....
Jüri Üyeleri :	**Yrd. Doç. Dr. Feza BUZLUCA** İstanbul Teknik Üniversitesi	.....
	Prof. Dr. Oya KALIPSIZ Yıldız Teknik Üniversitesi	.....

Teslim Tarihi : **17 Mayıs 2012**
Savunma Tarihi : **3 Ağustos 2012**

ÖNSÖZ

Bu çalışma, İstanbul Teknik Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim dalı, Bilgisayar Mühendisliği Programında, "Simülasyon Yazılımlarında Kod Klonları" konulu yüksek lisans tezi olarak hazırlanmıştır.
Çalışmanın başlangıç aşamasından bitimine dek, aydınlatıcı yorumları ve katkılarından dolayı danışman hocam Prof. Dr. Muhittin GÖKMEN'e,
Tezin çeşitli aşamalarındaki değerli eleştirileri ve önerilerinden dolayı Yrd. Doç Dr. Hasan SÖZER'e,
Tez için gerekli kod tabanının paylaşımını sağlayan ve analiz sürecinde desteğini esirgemeyen TÜBİTAK BİLGEM bünyesindeki yazılım mühendislerine ve yöneticilerine,
Tüm eğitim hayatım boyunca, özveri ve desteği ile hep yanımda olan değerli aileme teşekkürü bir borç bilirim.

Ağustos 2012

Merve ASTEKİN
(Bilgisayar Mühendisi)

İÇİNDEKİLER

Sayfa

ÖNSÖZ ..... **v**
İÇİNDEKİLER ..... **vii**
KISALTMALAR ..... **ix**
ÇİZELGE LİSTESİ ..... **xi**
ŞEKİL LİSTESİ ..... **xiii**
ÖZET ..... **xv**
SUMMARY ..... **xvii**
1. GİRİŞ **1**
1.1 Tezin Amacı 1
1.2 Literatür Araştırması 2
1.3 Hipotez 3
2. YAZILIM KOD KLONLARI **5**
2.1 Klon Terminolojisi 5
2.1.1 Klon tipleri 5
2.2 Klonlamanın Nedenleri 7
2.3 Klonlamanın Yararları ve Riskleri 8
2.4 Klon Belirlemenin Uygulama Alanları 9
2.5 Klon Belirleme Teknik ve Araçları 10
2.5.1 Metin-tabanlı yaklaşım 10
2.5.2 Simge-tabanlı yaklaşım 11
2.5.3 Metrik-tabanlı yaklaşım 11
2.5.4 Soyut sözdizim ağacı-tabanlı yaklaşım 12
2.5.5 Program bağımlılık grafı-tabanlı yaklaşım 13
2.5.6 Melez yaklaşımlar 13
2.6 Klon Belirleme Algoritmalarının Karşılaştırılması 14
3. KULLANILAN TEKNİK VE ARAÇ **17**
3.1 Klon Tespit Aracı Özellikleri 17
3.2 Kod Klonu Tespit Süreci 19
3.2.1 Sözcüksel analiz 19
3.2.2 Dönüşüm 19
3.2.3 Eşleşen belirleme 20
3.2.4 Formatlama 21
4. ÇALIŞMA ALANI: SİMÜLASYON SİSTEMLERİ **23**
5. ANALİZ SÜRECİ VE DENEYSEL SONUÇLAR **25**
5.1 Analiz Ortamı 25
5.2 Sistemlerin Kendi İçlerindeki Klonlar 26
5.3 Sistemler Arası Klonlar 29
5.4 Referans Mimari Tanımlama/Detaylandırma 34
6. SONUÇLAR VE GELECEK ÇALIŞMA **37**
KAYNAKLAR **39**
ÖZGEÇMİŞ **43**

KISALTMALAR

AST : Abstract Syntax Tree
BİLGEM : Bilişim ve Bilgi Güvenliği İleri Teknolojiler Araştırma Merkezi
BTE : Bilişim Teknolojileri Enstitüsü
FOM : Federation Object Model
HLA : High Level Architecture
PDG : Program Dependency Graph
RTI : RunTime Infrastructure
TÜBİTAK : Türkiye Bilimsel ve Teknolojik Araştırma Kurumu
XML : Extensible Markup Language

ÇİZELGE LİSTESİ

Sayfa

Çizelge 2.1 : Bellon'a ait Deney Kapsamı 15
Çizelge 2.2 : Bellon'a ait Deney Sonuçları 15
Çizelge 2.3 : Rysselberghe ve Demeyer'a ait Deney Sonuçları 16
Çizelge 4.1 : İncelenen Sistemlerin Boyut Bilgileri 23

ŞEKİL LİSTESİ

Sayfa

Şekil 2.1 : Tip 1 Klonu. 6
Şekil 2.2 : Tip 2 Klonu. 6
Şekil 2.3 : Tip 3 Klonu. 6
Şekil 2.4 : Tip 4 Klonu. 7
Şekil 3.1 : CCFinder simge tabanlı klon tespit işlemi. 19
Şekil 3.2 : Dönüşüm kurallarının uygulanması. 20
Şekil 3.3 : Parametre yer değişimi ile dönüşüm işlemi. 20
Şekil 5.1 : Analiz Süreleri..... 25
Şekil 5.2 : Sistemlerin kendi içlerinde gerçekleştirilen analiz süreci 26
Şekil 5.3 : Sistemlerin kendi içlerinde klon dağılımları (CLN)..... 27
Şekil 5.4 : Sistemlerin kendi içlerinde klonların dosyalara saçılımı (NIF) 28
Şekil 5.5 : Proje Büyüklüğü – Klon İlişkisi..... 29
Şekil 5.6 : Sistemler arasında ikili gerçekleştirilen analiz süreci 30
Şekil 5.7 : Kesişen klonların (X-Z) her bir sistemdeki dağılımı 31
Şekil 5.8 : Kesişen klonların (Y-Z) her bir sistemdeki dağılımı 32
Şekil 5.9 : Kesişen klonların (Z-T) her bir sistemdeki dağılımı 33
Şekil 5.10 : Tanımlanan tekrar kullanılabilir bileşenler ile referans mimari..... 34

SİMÜLASYON YAZILIMLARINDA KOD KLONLARI

ÖZET

Son yıllarda, yazılım dünyasındaki hızlı gelişim geniş ölçekli yazılım projelerini beraberinde getirmiştir. Boyutları hızla artan yazılım sistemlerinde, model ve kod kalitesini sağlayabilmek güçleşmiştir. Büyük yazılım sistemlerinde sıkça karşılaşılan, model ve kod kalitesini önemli ölçüde etkileyen bir parametre, bilinçli ya da bilinç dışı kullanım sonucu oluşan kod tekrarlarıdır.

Bazı benzerlik tanımlarına ya da fonksiyonlarına göre benzer olan kod parçaları olarak tanımlanan kod klonları, açık kaynaklı veya endüstriyel yazılım sistemlerinde sıklıkla gözlenmektedir. Geliştiriciler tarafından benimsenen geliştirme tekniği olarak kod parçalarının küçük değişiklikler ile ya da aynen tekrar kullanımı, kod klonlarına sebep olmaktadır.

Kod klonlarının yazılım sistemlerine zararları ve bazı açılardan yararları olduğu gerekçeleriyle, sistem için avantajlı ya da dezavantajlı oldukları kararı netleştirilememiş, açık bir argümandır. Sistem bakımı, yeniden düzenlemesi, hata yayılımı, tasarım kalitesi, yeniden kullanılabilirlik, performans gereksinimleri, kod anlaşılırlığı ve karmaşıklığı yönleriyle, yazılım sistemlerine zararlı oldukları düşüncesi hakimdir.

Sistem üzerindeki olumsuz etkileri nedeniyle varlığı istenmeyen kod klonlarının, günümüz geniş ölçekli yazılım sistemlerinde manuel olarak tutarlı bir şekilde kaldırılması mümkün olmamaktadır. Otomatize bir yöntem ile tespit edilmesi zorunlu hale gelen bu kavramın saptanması için, literatürde birçok teknik ve araç önerilmektedir.

Günümüze dek, otomatik kod klon tespiti için birçok teknik ve araç önerilmiş ve değerlendirilmiştir. Bu teknikler genel olarak, metin-tabanlı, simge-tabanlı, soyut sözdizim ağacı-tabanlı, program bağımlılık grafı-tabanlı teknikler ile tüm bu teknikler birleştirici nitelikteki melez yaklaşımlar sınıflandırılırlar.

Klon tespit teknikleri ve araçları üzerine aktif birçok çalışma olsa da, çeşitli sistemlerdeki klonlama davranışını kapsamlı olarak ele alan çalışmalara sıklıkla rastlamak mümkün değildir. Klonlama üzerine gerçekleştirilen deneysel çalışmaların odağı genellikle teknikleri karşılaştırmak, geçerlemek iken, bu çalışmalar genellikle, özel sistemler üzerine klonlama karakteristiklerini çalışma amacı olarak ele almamaktadırlar.

Otomatik klon tespiti için önerilen teknikler, web uygulamaları ve cihaz sürücüleri başta olmak üzere birçok farklı alanda açık kaynaklı sistemler üzerinde uygulanmıştır. Klon tespit teknikleri, ayrıca uygulama alanına dair kavramların tanımlanmasında ve yazılım üretim bandının iyileştirilmesinde de kullanılmaktadır. Bu yaklaşım, endüstriyel durum çalışmalarında şimdiye kadar denenmemiştir.

Bu çalışma, klon tespit yönteminin etkinliğini alan analizi yönüyle incelemektedir. Çalışmada, klon tespit aracı olarak CCFinder kullanılmıştır. CCFinder klon tespit aracı, simge-tabanlı algoritması ve kullandığı birçok optimizasyon tekniği ile geniş ölçekli yazılım sistemleri için verimli ve ölçeklenebilir bir çözüm niteliği taşımaktadır.

Çalışmada, simge-tabanlı klon tespit yönteminden yararlanıldı. Simge-tabanlı teknik, sözdizimsel eş kopyalar olan Tip 2 klonlarının tespiti için yüksek başarıma sahiptir. Bu yaklaşımda, tüm kaynak sistem bütünüyle simge sekansına dönüştürülür. Klonlar, bu sekansın tekrar eden alt simge sekanslarını bulmak üzere taranması ile tespit edilirler.

Çalışmada, dört farklı endüstriyel simülasyon yazılım sistemi incelenmiş ve değerlendirilmiştir. Simülasyon sistemlerinden kimileri aynı mimariyi kullanmakta ya da benzer takımlar tarafından geliştirilir iken, kimileri bağımsız olarak geliştirilmiştir. Birbirinden farklı büyüklüklerdeki bu sistemler, analiz için uygun çeşitliliği sağlamaktadırlar.

Klon tespit yönteminin simülasyon sistemleri alan analizi için uygulanabilirliğini araştıran bir çalışma gerçekleştirilmiştir. Bu alanda dört farklı simülasyon projesi incelenmiş, uygulama alanı kavramlarının bir kümesi ile tekrar kullanılabilir bileşenler tanımlanmıştır. Bu doğrultuda, yüksek düzeyli mimariyi temel alan bir referans mimari tanımlanmıştır. Yaklaşımın verimliliğini nicel olarak ölçebilmek için tanımlanmış ve kesinleştirilmiş klon kümesi ya da tekrar kullanılabilir bileşenlerin tam ve kesin bir listesi gibi herhangi bir dayanak bulunmamaktadır. Analiz ile elde edilen sonuçların doğruluğu ve yararlılığı, uygulama alanı uzmanları ve yazılım mimarları tarafından onaylanmıştır. Klon tespit yönteminin, alan analizi ve referans mimari tanımlama/detaylandırma için uygulanabilir olduğu görülmüştür. Ayrıca, tanımlanan tekrar kullanılabilir bileşenler ile, simülasyon projelerinin tamamı için yararlı olabilecek "Yazılım Ürün Hattı" aday bileşenleri belirlendi.

Çalışma, klon büyüklüğünü, dağılımını ve yoğunluğunu sistemlerin kendi içlerinde ve birbirleri ile aralarında incelemektedir. Çalışma sürecinde gözlenen diğer sonuçlar olarak, projedeki kod satır sayısının/dosya sayısının fazla olmasının, aynı mimari altyapıya sahip olmanın, benzer geliştirme ekipleri tarafından geliştirilmenin ve benzer alan hizmeti sunmanın klon sayısını artıran etkenler olduğu görülmüştür.

Projelerin kendi içlerinde gerçekleştirilen analizler ile, yazılım kalitesini artıran bakım işlemi kapsamında yeniden düzenleme çalışmalarının yoğunlaşacağı ve yüksek hata olasılığına sahip alanlara referans oluşturuldu. Ayrıca, projelerin kendi içlerindeki klonların iyi kullanılamamış ya da kullanılması gerektiği halde kullanılmamış kalıtım yapıları nedeniyle de oluştuğu görülmüştür. Bu durum, sınıf yapıları kararlarının verildiği tasarım aşamasına bağlanarak, yazılım kalitesi özelliği olarak değerlendirilen iyi kullanılmış kalıtım özelliğinin kaliteli bir tasarımdan üretilebileceğini destekler niteliktedir.

CODE CLONES IN SIMULATION SOFTWARE SYSTEMS

SUMMARY

In recent years, rapid development of software's world has brought with large scale software projects. Increasing size of growing software systems made difficult to provide and ensure model and code quality. Reusing code as a result of concious or unconcious use is a parameter which is frequently encountered in large scale software systems and significantly affects quality of model and code.

Code clones are defined as the segments of code which are similar according to some definition of similarity. They typically result from reusing code fragments with or without minor modifications. They can be often observed in both open source and industrial software systems.

This is an open argument and indeterminate issue that code cloning is a detriment to software systems. Pros and cons of the effects of cloning on systems should be weighed by examination with respect to diverse perspectives including system maintenance, refactoring, bug propagation, design/reusability, performance requirements, code understandability. According to these views, cloning is mainly considered as harmful for software systems.

Detection and removal of code clones is desired because of effects on software systems considered as detrimental. Increasing complexity and number of projects makes it costly to manually analyze commonality/variability among different systems. Therefore, an automated way for this operation is essential and detection of code clones has been an active research area over the last decades.

Automated clone detection is an active field of research, and up to date, many techniques and tools have been proposed and evaluated for detection of different types of clones. While, Type 1 is an exact copy without modifications (except for whitespace and comments), Type 2 is a syntactically identical copy; changes only on variable, type, or function identifiers. Type 3 is a copy with modified/removed/added statements. Type 4 indicates functional similarity, and includes two or more code fragments that perform the same computation with different syntactic variants.

The techniques behind clone detection tools are distinguished according to the type of information their analysis based on and the used algorithms. The techniques based on text, token, metric, abstract syntax trees (AST), program dependency graph (PDG) comparison, and the other hybrid techniques that combine specialized comparison.

Textual Comparison is based on textual comparison of whole lines or efficient string matching based on fingerprints. Token Comparison requires the entire source code to be transformed into a sequence of tokens. Duplicated subsequences of tokens are detected thereafter. Metric Comparison relies on a set of metrics defined to compare different code fragments with each other. Abstract Syntax Tree Comparison works on parse trees or abstract syntax trees (ASTs) to find similar sub-trees as clones.

Program Dependency Graph Comparison is based on using a program dependency graph. The nodes of this graph represent expressions and statements. The edges symbolize control and data dependencies. Other Hybrid Techniques combine different approaches to achieve better accuracy.

Many clone detection techniques and tools have been proposed using various techniques and algorithms. These techniques have been mostly applied on open source software to analyze cloning in different domains and systems such as web applications and device drivers. Clone detection techniques have also been proposed for identifying domain concepts and enhancing software product line development. However, the effectiveness of this approach has not been evaluated yet in the context of industrial case studies.

This study investigates the effectiveness of clone detection in supporting domain analysis. For the analysis, the CCFinder clone detection tool is used and evaluated on four industrial simulation software systems.

In this study, a token-based code clone detection tool has been used, CCFinder. The reason for choosing this tool is that it has a precision comparable to the other techniques and yet, it has the highest recall. The tool is also pretty suitable, efficient, scalable for large scale software systems with developed optimization techniques in tool.

The token-based clone detection is utitlized in this study. The technique is highly succesful for Type 2 clones which are syntactically identical copies. The token-based clone detection process in CCFinder consists of four basic steps: Lexical Analysis, Transformation, Match Detection, and Formatting. In Lexical Analysis step, each line of source code is divided into a set of tokens. In Transformation step, a set of transformation rules are applied for regularizing identifiers, and identifying structures. As such, clone detection becomes agnostic to variable names. Mapping information between the transformed and the original token sequences is stored. In Match Detection step, equivalent pairs (i.e., clones) for all the sub-strings in the token sequence are detected. In Formatting step, mapping information is used for locating the detected clones in the source code files.

CCFinder uses several parameters including minimum clone length (MCL) and minimum TKS shaper level. The parameter settings (MCL=100, min. TKS=10) that are suggested for the analysis of large software systems are used in the study.

CCFinder presents a metric set which points to interesting clones. Some of these metrics are used in the analysis for identifying systems' cloning behavior. *CLN* which represents accumulation of clones per each source code file, *NIF* which measures the amount of scattering of each code clone among the source code files and *RAD* which defines the range of source code fragments regarding a code clone in a directory hierarchy are selected for the analysis.

Four different simulation systems have been studied in this study. Simulation software is widely used in many application domains to investigate real-world conditions that are difficult or costly to experiment with. TUBITAK has years of experience in developing simulation software for different type of systems (e.g., trains, submarines). The size and complexity of simulation software increases because of the complexity of the simulated systems. Systematic software reuse at the architectural level becomes essential to increase software quality and decrease the development time and costs. However, increasing complexity and number of projects

makes it costly to manually analyze commonality/variability among different systems. In this study, the effectiveness of a clone detection tool has been investigated in supporting such a domain analysis. Due to confidentiality, detailed information can not be presented regarding the corresponding projects. In the rest of the paper, these projects will be referred as Project X, Project Y, Project Z, and Project T. Some of these systems share a common architectural style or they have been developed by a common team, whereas some of them have been developed independently. These projects which are at quitely different sizes are appropriate for analysis with their variety.

Project Z and Project T adopt the same architectural style based on High Level Architecture (HLA). Also, the de-velopment team of Project T is a subgroup of the development team of Project Z. Project X has a separate development team, which does not follow HLA rules but adopts some of the features of HLA. Project Y has also a separate, independent development team. This project uses a simulation engine infrastructure designed for a particular application domain and it does not employ HLA. These projects have been analyzed with CCFinder to identify commonalities.

A case study on utilizing clone detection for domain analysis of simulation systems have been conducted. Four industrial software systems in this domain have been analyzed. The clone size, distribution and density both within each system and across the four systems have been examined. A set of domain concepts and reusable components have been identified. Accordingly, a reference architecture based on HLA have been defined. The metrics for the (quantitative) measure of effectiveness of the approach do not exist. However, the validity and usefulness of the results were confirmed by the domain experts and software architects. As such, utilization of clone detection can be a viable approach for supporting domain analysis and definition/refinement of a reference architecture. Besides, software product line candidate components which are useful for the simulation software systems are identified with defined reusable components in this study.

The clone size, distribution and density both within each system and across the four systems are examined in the study. As a result of the analysis, increased line of code or number of files of the projects, having similar architectural infrastructure, being developed by similar development team and presenting similar domian service increases cloning ratio in systems.

The results of study are reference for selection of points which are focus for maintenance and refactoring processes, and also highly bugy regions, by means of the analysis within each system. Moreover, it has been experienced that one other reason of cloning is code clones within projects, which have no or bad inheritance structure or abstraction. Quality of inheritance structure is highly dependent on the quality of design phase. In this way, it is concluded that design is a quite important factor on cloning ratio of software systems.

1. GİRİŞ

Son yıllarda, yazılım dünyasındaki hızlı gelişim geniş ölçekli yazılım projelerini beraberinde getirmiştir. Boyutları hızla artan yazılım sistemlerinde, model ve kod kalitesini sağlayabilmek güçleşmiştir. Büyük yazılım sistemlerinde sıkça karşılaşılan, model ve kod kalitesini önemli ölçüde etkileyen bir parametre, bilinçli ya da bilinç dışı kullanım sonucu oluşan kod tekrarlarıdır.

Kod klonları, bazı benzerlik tanımlarına göre benzer olan kod parçaları olarak tanımlanmaktadırlar [1]. Tipik olarak, kod parçalarının küçük modifikasyonlar ile ya da hiçbir değişiklik yapmadan tekrar kullanımı sonucu oluşmaktadırlar.

Kod klonlamanın bazı sistemler üzerinde avantajlı olduğu yönler var ise de [2], yazılım sistemlerine zararlı olduğu konular mevcuttur. Sistem bakımı, yeniden düzenlemesi, hata yayılımı, tasarım, yeniden kullanılabilirlik, performans gereksinimleri, kod anlaşılırlığı ve karmaşıklık, yazılım klonlarının zararlı olduğu konulara örnek olarak verilebilir.

Kod klonlarının tespiti, son yıllarda aktif bir çalışma alanı haline gelmiş olup, bu konuda birçok teknik ve araç önerilmiştir [3]. Metin-tabanlı, simge-tabanlı, soyut sözdizim ağacı-tabanlı, program-bağımlılık grafı teknikleri ile bu teknikleri birleştiren nitelikte melez teknikler mevcuttur.

Önerilen klon tespit teknik ve araçları, web uygulamaları [4] ya da cihaz sürücüleri [5] gibi farklı alan ve sistemler üzerinde klonlama davranışını analiz etmek üzere çoğunlukla açık kaynaklı yazılımlara uygulanmışlardır. Ayrıca, klon tespit teknikleri, alan kavramlarını tanımlamak, bu yolla, yazılım ürün hattı gelişimi iyileştirmek için önerilmişse de [6], bu yaklaşımın etkinliği, endüstriyel durum çalışmaları üzerinde henüz değerlendirilmemiştir.

1.1 Tezin Amacı

Çalışma, klon tespit yönteminin alan analizinde kullanılabilirliğini ve etkinliğini araştırmaktadır. İncelenen simülasyon sistemlerinin kendi içlerinde ve birbirleri ile

aralarındaki kod klonları incelenerek ortak bileşen analizi gerçekleştirmektedir. Analiz sonuçları ile alan analizi, yeniden kullanılabilir bileşen tanımlaması ile yazılım ürün hattı geliştirilmesi ve referans mimari tasarımı gerçekleştirilmesi amaçlanmıştır.

1.2 Literatür Araştırması

Günümüze dek, yazılım kod klonları üzerine araştırmalar çoğunlukla yeni ve verimli klon tespit teknikleri, araçları ve algoritmaları üzerine yoğunlaşmışlardır. Kod klonları üzerine çok sayıda deneysel çalışma mevcut olsa da, bu çalışmaların temel amacı, önerilen tekniklerin kod klon tespitindeki başarımını test etmektir.

Yakın zamanlarda yapılan çalışmalar, özel sistemler üzerinde klonlama davranışını analiz etmeye başlamışlardır. Özel sistemler, proje yaşam döngüsü sürecince klonlama oranı incelemesi yaparak klon evrimi üzerine elde edilen sonuçları tartışan klonların evrimi [7] yönleriyle alınmışlardır. Ayrıca, web uygulamalarındaki klonları inceleyen kapsamlı bir çalışma mevcuttur [4].

Basit ve diğerleri, farklı boyutlardaki 17 web uygulaması üzerinde sistematik bir çalışma gerçekleştirmişler ve web mühendisliğinin önemli problemlerine tekrar kullanım temelli yöntemlerin incelenmesi ile çözüm öneren sonuçlar üretmişlerdir [4].

Wang ve Godfrey klon evrimine odaklanarak, farklı mimari katmanlarda 16 yıllık bir evrim sürecine yayılan Linux işletim sistemi için SCSI sürücüleri arasındaki klonlar üzerine kapsamlı bir çalışmanın sonuçlarını sunmuşlardır [7].

Roy ve Cordy, açık kaynaklı yazılım sistemlerinde işlev klonları üzerine ilk deneysel çalışmayı sunmuşlardır [8]. Çalışmayı, yazılımlardaki klonlanmış kod tanımlamasındaki başarımı artırmak amacıyla geliştirdikleri metin-tabanlı ve soyut sözdizim ağacı-tabanlı tekniklerin tespit başarımını birleştirip kısıtlamalarını gideren melez klon tespit aracı NICAD ile gerçekleştirmişlerdir. Linux Kernel ve Apache httpd sistemlerini de içeren 15'den fazla açık kaynaklı C ve Java projesi üzerince gerçekleştirilen çalışmada, klonlanmış kod kullanımı birçok farklı açıdan ele alınmıştır. Dil, klon büyüklüğü, klon yerleşimi, klonlanmış fonksiyondaki klon yoğunluğu gibi konular, çalışmada ele alınan yönlerdir.

Özelliğe dayalı yazılım üretim hattında kod klonları, farklı birçok alan için incelenmiştir [9]. Yazılım varyantları arasındaki ortak bileşenleri tanımlamak için klon tespiti ele alınmıştır [10].

Ira Baxter tarafından başlatılan çalışmada, yazılım ürün hattı gelişimini iyileştirme amaçlı klon tespit yöntemine yoğunlaşılmıştır [6]. Klonlardaki hata tespiti, artık kodların kaldırılması, mevcut ya da gelecek sistemlerdeki kullanımı için alan kavramlarının tanımlanması, sistem bakım maliyetinin düşürülmesi, yeniden kullanılabilir bileşenlerin tespiti, çalışmada ele alınan konular olmak ile birlikte, çalışmanın önerdiği yaklaşım uygulamaya geçirilmemiştir.

Çalışmanın karakteristiğine yakın olarak, cihaz sürücülerinin alan analizi çalışması, klon tespit yöntemi kullanılarak gerçekleştirilmiştir [6]. CCFinder klon tespit aracı kullanılarak, açık kaynaklı 13 cihaz sürücüsünün ortak davranışları analiz edilmiştir. Çalışmanın sonuçları, aynı alana ait cihaz sürücüleri arasında oldukça fazla sayıda kod klonu bulunduğunu göstermiştir. Bu yaklaşım henüz, endüstriyel durum çalışması üzerinde gerçekleştirilmemiştir.

1.3 Hipotez

Yazılım kod klonu tespit yöntemi kullanılarak açık kaynaklı yazılım sistemleri üzerinde gerçekleştirilen alan analizi çalışması, endüstriyel sistem olan simülasyon yazılımları üzerinde denenmemiştir. Bu çalışma, klon tespit yöntemi ile simülasyon sistemlerine ait ortak bileşenlerin tanımlanabileceğini göstermektedir. Analiz sonuçları, alan analizini, yeniden kullanılabilir bileşenlerin tanımlanmasını ve referans mimari tasarımını destekler niteliktedir.

2. YAZILIM KOD KLONLARI

Bu bölümde yazılım kod klonlarına ait bilgi verilecektir.

2.1 Klon Terminolojisi

Yazılım kod klonları alanında önemli bu bölümde açıklanacaktır.

Kod Klonu: Kod klonları, bazı benzerlik tanımlarına göre benzer olan kod parçaları olarak tanımlanmaktadır [1]. Metin, sözdizimi, sözcük, anlam ya da örüntü tabanlı gibi benzerlik tanımları bulunmaktadır [11].

Klon Çifti: Tamamıyla aynı ya da birbirlerine benzer olabilecek bir klon ilişkisine sahip kod parçaları çiftlerine klon çifti adı verilmektedir [2].

Klon Sınıfı: Kod parçalarının herhangi bir çifti arasında bir klon ilişkisinin olduğu kod parçalarının en büyük kümesidir [2]. Ortak kod bölümlerine sahip olan tüm klon çiftlerinin birleşimi olarak ifade edilmektedir [12].

Klon Sınıfı Ailesi: Kaynak kod parçalarının geldiği dosya, sınıf, fonksiyon ya da paket gibi kaynak varlıklar kümesi olarak ifade edilen alanları aynı olan tüm klon sınıflarının kümesine klon sınıfı ailesi ya da süper klon adı verilmektedir [2]. Aynı kaynak varlıklar arasındaki birden çok klon sınıfı, bir süper klon içerisinde birleştirilirler.

2.1.1 Klon tipleri

İki kod parçası arasında olabilecek temel olarak iki tür benzerlik bulunmaktadır. Bu benzerlik türlerinden biri, program metninin benzerliği; diğeri ise, metin benzerliğinden bağımsız, işlevsellikteki benzerliktir.

Metinsel benzerlik aşağıdaki klon tiplerine ayrılmaktadır [13]:

- **Tip 1:** Boşluk karakterleri ve yorum ifadeleri dışında herhangi bir modifikasyona tabi tutulmayan tam kopyadır. Şekil 2.1'de, sol tarafta verilen kod parçasına ait Tip 1 klonu, ilgili resmin sağ tarafında görülebilir:

```
if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;}
else
    c = d - a; //Comment2
```

```
if (a>=b) {
     // Comment1'
    c=d+b;
    d=d+1;}
else  // Comment2'
    c=d-a;
```

Şekil 2.1 : Tip 1 Klonu.

Şekil 2.1'de, boşluk karakterleri ve yorum satırlarının kaldırılması sonrası, her iki kod parçasının birbiriyle özdeş olduğu görülebilir.

- **Tip 2:** Yapısal ve sözdizimsel olarak birbirlerinin özdeş kopyası olan klonlardır. Yalnızca değişken, tip ve fonksiyon tanımlayıcılarında değişiklikler içerirler.

```
if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;}
else
    c = d - a; //Comment2
```

```
if (m >= n)
    { // Comment1'
    y = x + n;
    x = x + 5; //Comment3
    }
else
    y = x - m; //Comment2'
```

Şekil 2.2 : Tip 2 Klonu.

Şekil 2.2'de, şekillerdeki, değişken isimlerindeki ve değer atamalarındaki değişikliklere rağmen, iki kod parçasının sözdizimsel olarak benzer oldukları görülebilmektedir.

- **Tip 3:** İfadelerdeki değişimler, ekleme ve çıkarmalar gibi modifikasyonlar içeren kopyalardır.

```
if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;}
else
    c = d - a; //Comment2
```

```
if (a >= b) {
    c = d + b; // Comment1
    e = 1; // This statement is added
    d = d + 1; }
else
    c = d - a; //Comment2
```

Şekil 2.3 : Tip 3 Klonu.

Şekil 2.3'de, kod parçalarının yalnızca "e = 1" ifadesi yönüyle farklılık kazandıkları görülebilmektedir.

İşlevsel benzerlik, Tip 4 klonlarında görülebilmektedir [2].

- **Tip 4:** Farklı sözdizimsel değişkenler ile aynı hesaplamayı gerçekleştiren iki ya da daha fazla kod parçasıdır.

```
int i, j=1;
  for (i=1; i<=VALUE; i++)
       j=j*i;
```

```
int factorial(int n) {
 if (n == 0) return 1 ;
 else        return n * factorial(n-1) ;
}
```

Şekil 2.4 : Tip 4 Klonu.

Şekil 2.4'de görülen iki kod parçasından biri basit bir kod parçası, diğeri özyinelemeli bir fonksiyon iken, her iki kod parçasının anlamsal açıdan benzer işlevselliğe sahip oldukları görülmektedir. Her iki kod parçası arasında, sözcüksel/sözdizimsel/yapısal herhangi bir benzerlik bulunmamaktadır.

2.2 Klonlamanın Nedenleri

Geliştirme stratejisi olarak klonlama, en önemli ve ana sebeplerden biridir [13]. Basit kopyala&yapıştır tekrar kullanımının yanı sıra, çatallanma ve tasarım, fonksiyonellik, mantık tekrar kullanımı ile tekrar kullanım yaklaşımları mevcuttur. Ayrıca, benzer sistemleri birleştirme, üretici programlama yaklaşımını kullanarak sistem geliştirme ve tekrar yapılandırma işlemlerinde yaşanan gecikmeler, programlama yaklaşımı yönüyle klonlara sebep olmaktadır. Geliştiricilerin program metnini şablon olarak kullanıp, bunun üzerinden değişiklikler yapıp ihtiyaçlara göre düzenlemesi de klon oluşumunun temel sebeplerinden biridir [14].

Geliştiriciler sistem bakımında sağladığı yararlar nedeniyle klonlamayı tercih edebilmektedirler [2]. Sıklıkla güncelleme ve iyileştirme gerektiren sistemlere destek sağlama amacıyla yürütülen geliştirme faaliyetlerinde, yeni kod geliştirme ile oluşabilecek riskler göz önüne alınarak, eski kodların tekrar kullanımı ile geliştirme yapılabilmektedir. Ayrıca, hayati kritiklik seviyesindeki sistemlerin gürbüzlüğünden emin olmak amacıyla klonlama tercih edilebilmektedir. Bu tür sistemlerde, aynı şartlar altında uygulamaların başarısız olma olasılığını azaltmak için klonlama kullanılabilmektedir. Bunun yanı sıra, gerçek zamanlı uygulamalarda, fonksiyon çağrılarının getirdiği maliyeti azaltmak amacıyla, aynı fonksiyonelliği sağlayan kod parçaları tekrar kullanılagelmektedir. Yalnızca değişen kod parçalarını test etme

yönüyle bakım işlemlerini hızlandırma, anlaşılabilir yazılım mimarisi sağlama amaçları ile de klonlama kullanılabilmektedir.

Dilden ya da geliştiriciden kaynaklanan kısıtlamaların üstesinden gelmek için klonlama tercih edilebilmektedir [13]. Programlama dilindeki soyutlama gibi tekrar kullanım mekanizmalarının eksikliği, dilde tekrar kullanılabilir kod yazmanın zor olması ve hataya açık olması gibi durumlar programlama dili tarafından getirilen kısıtlara örnek olarak verilebilir. Geliştiricinin geniş ölçekli sistemleri anlamada güçlük çekmesi, performansının değerlendirilmesi için kullanılan kod satır sayısı ve zaman kısıtı ölçekleri, problem alanına ait bilgi eksikliği ve tekrar kullanılabilir kod yazma sorumluluğununun eksikliği ise geliştiriciden kaynaklanan klonlama nedenleri olarak gösterilmektedir [14].

Klonlamanın bir diğer nedeni ise hata ile oluşan klonlardır [13]. Kütüphane ile yazılım programlama arayüzleri arasındaki etkileşim protokolleri, farklı geliştiriciler tarafından farkında olmadan aynı mantığın geliştirilmesi, geliştiricinin önceki deneyimlerinden edindiği kodlama davranışı, hata ile klon oluşturmaya neden olmaktadır.

2.3 Klonlamanın Yararları ve Riskleri

Klonlamanın yazılım sistemleri üzerindeki etkisinin olumlu ya da olumsuz olduğuna dair farklı değerlendirmeler mevcuttur [2]. Bu yönüyle, yazılım kod klonlarının yararlı ya da zararlı oldukları konusu açık bir argüman olarak varlığını sürdürmektedir.

Klonlamanın etkileri, sistem değişimi üzerindeki etkisi, hatalar üzerindeki etkisi, bilişsel efor üzerindeki etkisi, tasarım üzerindeki etkisi ve yeniden düzenleme üzerindeki etkisi yönleriyle ele alınabilir [13].

Klonlamanın sistem değişimi üzerindeki etkisi, klonlanmış uygulamayı ve uyarlanacak kısımları anlamada ek süre ve dikkat gerekmesi yönüyle ele alınabilir. Ayrıca, bir kod parçasını değiştirmenin tüm klonlarını da değiştirmeyi gerektirmesi ve klonlar üzerindeki değişimlerin tutarsız bir şekilde yapılması, bakım yapılacak ve iyileştirilecek kod parçaları için iş gücünün çoklanmasına neden olur. Büyük ölçekli sistemler için kopyalanmış kod parçalarının manuel aranması, diğer kod parçaları ile karşılaştırılması ile oluşacak detaylı analiz süreci, oldukça maliyetlidir [14].

Klonlama, hata yayılım olasılığını artırır ve kopyalanmış, tutarsız tekrar isimlendirilerek yapıştırılmış kodlar oluşmasına neden olur. Anlık ihtiyaçları karşılamak için değiştirilen kopyalanmış kod parçaları, yeni hatalar oluşmasına sebep olabilir [1, 15].

Klonlama, ayrıca geniş ölçekli sistemlerde bakım için gereken bilişsel efor artışına neden olur iken, kalite yönüyle de iyi tasarlanmış kalıtım ve soyutlama yapısı eksikliğine neden olmaktadır. Bu şekilde oluşan kötü tasarım, gelecek projeler için yararlanılabilecek tekrar kullanılırlığı ve bakım yapılabilirliği azaltmaktadır [2].

Soyutlama mekanizması maliyetinin, klonlamanın dezavantajlarından yüksek olabilmesi, bazı kod parçalarının tekrar kullanımının bakım maliyetini azaltması ve yeni hata oluşumunu engellemesi yönleriyle yeniden düzenleme açısından etkisi gözlenmektedir.

2.4 Klon Belirlemenin Uygulama Alanları

Klon tespit çalışmasının çıktıları farklı birçok alanda kullanılabilmektedir. Kopyalanarak birçok defa kullanılan kod parçasının tekrar kullanılabilirliğini sistematik hale dönüştürmek üzere klon tespit yöntemi kullanılarak, kütüphane adaylarının belirlenmesi alanında kullanılmaktadır [16, 17].

Telif hakları ihlallerinin tespiti, yasadışı kopya yazılımların belirlenmesi alanlarında klon tespit yönteminden yararlanılmaktadır [2].

Yazılım bileşenlerinin bir sürümden diğer sürüme karşılaştırılmalarını ve böylece, farklı zamanlara/sürümlere göre değişimini esas alan yazılım sistemlerinin sürüm karşılaştırmalarında ve yazılım evrimi alanlarında da klon tespit yönteminden yararlanılmaktadır [14].

Zararlı yazılım adaylarının zararlı yazılım olduğu bilinen sistemler ile karşılaştırılması yoluyla zararlı yazılımların tespitinde, uygulamanın tümünde tekrarlı kopyalanmış olan enine kesen ilgilerin tespiti yönüyle ilgiye yönelik programlamada ilgilerin saptanmasında [18, 19], kaynak kod boyutunu azaltmak için kod sıkıştırmada [2], tekrar kullanılabilir yazılım bileşenlerinin tanımlanması yönüyle yazılım ürün hattı analizinde [13] klon tespit yöntemi kullanılmaktadır.

2.5 Klon Belirleme Teknik ve Araçları

Yazılım kod klonlarının manuel olarak bulunması, büyük ölçekli sistemlerde uygulanabilir olmamaktadır. Sistem boyutu arttıkça, bu işlem için otomatik bir yöntemi ihtiyacının artması klon tespit araçlarını beraberinde getirmiş ve bu alanı popüler bir araştırma alanı haline getirmiştir.

Klon tespit araçlarının arkasında kullanılan teknikler, kullandıkları algoritmalar ve analizlerinin dayandığı bilgi türüne göre ayrıştırılabilir [3]. Metinsel, simge, metrik, soyut sözdizim ağacı ve program bağımlılık grafı karşılaştırma ile bu karşılaştırma yöntemlerini birleştiren melez yaklaşımlar bulunmaktadır [14].

2.5.1 Metin-tabanlı yaklaşım

Metin-tabanlı yaklaşımda, hedef program satırlardan ya da katarlardan oluşan bir sekans olarak kabul edilir ve birbirinin aynısı olan metin ya da katar sekansları bulunmak üzere, kod parçaları karşılaştırılır [2].

Bu yöntemde, program metninin tüm satırlarının birbirleri ile metinsel karşılaştırılması [20] ya da parmak izlerine dayalı [21] etkili katar eşleme [22] yaklaşımları mevcuttur.

Johnson tarafından önerilen yaklaşımda [22], öncelikle pencere olarak adlandırılan belirli sayıdaki satır koda hash fonksiyonu uygulanır. Aynı hash değerine sahip olan satırları klon olarak belirleyebilmek amacıyla, artırımlı hash fonksiyonu ile kayan pencere tekniği uygulanır [13]. Çeşitli boyutlardaki kayan pencere ile, farklı boyutlardaki kod klonları bulunur.

Rieger tarafından önerilen yaklaşım, program metninin tüm satırlarının birbirleri ile metinsel karşılaştırılmasına dayalıdır [20]. Yüksek performans sağlayabilmek amacıyla, katarlar üzerinde uygulanan bir hash fonksiyonu sonucuna göre bölümlendirilen satırlardan, yalnızca aynı bölümde yer alan satırların karşılaştırılması ile işlem gerçekleştirilir. Sonuçlar noktasal grafik ile görselleştirilir.

Bu yöntemin temel avantajı, her programlama diline uygulanabilir olmasıdır. Diğer yandan, metindeki küçük değişiklikler analizin yanlış sonuçlanmasına neden olabilmektedir. Tip 1 klonlarını belirlemek için uygundur.

2.5.2 Simge-tabanlı yaklaşım

Bu karşılaştırma yönteminde, tüm kaynak sistem bütünüyle simge sekansına dönüştürülür. Klonlar, bu sekansın tekrar eden alt simge sekanslarını bulmak üzere taranması ile tespit edilirler.

Baker tarafından geliştirilen Dup isimli klon tespit aracı, satır temelli olup, satırdaki katarlar yerine simge sekanslarının son ek ağacı boyunca karşılaştırılması yöntemini kullanır [23]. Hash fonksiyonu olarak görev alan ve bir satırın karakter sekansının eşsiz olarak belirtimini sağlayan olan *functor* isimli yapı kullanılır. *Functor*, tanımlayıcı ve sözcüklerin somut değerlerini parametre olarak alarak özetler. Özetleme işlemini, parametrelerin sıralarından değil, somut değerlerinden gerçekleştirdiği için, parametrelere ait sistematic yeniden isimlendirme yapıldığı durumlarda, klon tespitini gerçekleştirebilmektedir [14].

Kamiya ve diğerleri, Baker tarafından önerilen tüm kaynak kodu sözcüksel simge sekansına dönüştüren bu tekniği, CCFinder klon tespit aracında genişletilerek küçük farklılıkları kaldırmak için yapılan ek kaynak normalizasyonları kullanmıştır [24]. Simge sekansları normalize edilerek tamlık değerini artırılmıştır. Çalışmada da kullanılan bu araç, Bölüm 3'te detaylı olarak açıklanmıştır.

Li ve diğerleri tarafından geliştirilen CP-Miner, diğer bir simge tabanlı gelişmiş klon ve klonlanmış kod nedenli hata tespit aracı olup, benzer kod parçalarını tespit işlemini veri madenciliği problemi olarak ele almaktadır [25]. CP-Miner simge-tabanlı yaklaşım kullanarak, sözcüksel bilgi ile ilgilenmeyen ve bu nedenle basit modifikasyonları tolere edemeyen katar-tabanlı yaklaşımın ve ayrıştırma ağaçları kullanarak yanlış pozitif üretme olasılığını arttıran sözdizimsel ağaç tabanlı yaklaşımın problemlerine karşı avantaj sağlamaktadır [25].

Simge-tabanlı teknik, metin-tabanlı yaklaşıma göre şekillendirme, boşluklandırma ve tekrar isimlendirme ile oluşan küçük kod değişikliklerine karşı daha gürbüzdür [2]. Tip 1 ve Tip 2 klonların tespiti için uygun bir yöntemdir.

2.5.3 Metrik-tabanlı yaklaşım

Metrik tabanlı yaklaşımlar genel olarak, kod parçaları için bir dizi metrik elde ederek, kod satırları ve soyut sözdizim ağacı yerine metrik vektörleri üzerinden karşılaştırma işlemini gerçekleştirmektedir.

Bu yaklaşımda, sınıf, fonksiyon, metot ya da ifade gibi sözdizimsel birimler için hesaplanan bir dizi yazılım metriği, temel alınan sözdizimsel birim üzerinden karşılaştırılarak kod klonları bulunmaya çalışılır [13].

Mayrand ve diğerleri, metrikleri programın her bir fonksiyon birimi için isimler, yerleşim, ifade ve kontrol akışlarından hesaplarken [26]; Patenaude ve diğerleri, McCabe çevrimsel karmaşıklığı, metottan yapılan çağrı sayısı, ifade sayısı gibi değerleri kullanarak metot düzeyinde hesaplamışlardır [27].

Markov modelleri kullanılarak soyut örüntü eşleştirimi ile programlar arası benzerliği ölçen [28] yaklaşımın yanı sıra, kod blokları için belirlenen özellik kümesi üzerinde sinirsel ağları kullanarak benzer kod parçalarını bulan [16] yaklaşımlar da bulunmaktadır. Ayrıca, web sayfaları bileşenleri arasındaki uzaklık ve benzerlik derecesi değerlerini kullanarak web dokümanları üzerinde tekrarlı web sayfaları ve klonlarının bulunmasını amaçlayan uygulaması [29, 30] mevcuttur.

Kolay ve hızlı olması avantajının yanı sıra, hatalı çıktı üretmeye neden olabilmektedir. Tip 1 ve Tip 2 klonları için uygundur.

2.5.4 Soyut sözdizim ağacı-tabanlı yaklaşım

Ağaç eşleme tekniğinin temel yapısı, kaynak program kodunu çözümleme ağaçlarına ve soyut sözdizimi ağaçlarına dönüştürme, ardından benzer alt ağaçları klon olarak tanımlama işlemidir. Değişken isimlerinin, sözcük değerleri ve ağaçtaki diğer yapraklara karşılık gelen diğer simgelerin soyutlanması ile gelişmiş bir saptama işlemine olanak sağlamaktadır [13].

Soyut sözdizim ağacı (Abstract Syntax Tree - AST) tekniğinin öncülerinden olan Baxter ve diğerlerine ait CloneDR aracı, program soyut sözdizim ağacını bir hash fonksiyonuna göre bölümlendirerek, ağaçta aynı bölümde bulunan alt ağaçları karşılaştırma işlemini gerçekleştirir [1]. Karşılaştırma sonucu benzer olduğu tespit edilen alt ağaçları klon olarak tespit eder.

Yang tarafından önerilen, dinamik programlama kullanarak, aynı dosyaya ait farklı sürümler arasındaki sözdizimsel değişiklikleri saptamaya dayalı yaklaşımın yanı sıra, kaynak kodun soyut sözdizimsel ağacının Genişletilebilir İşaretleme Dili (XML) dönüşümü üzerinde veri madenciliği teknikleri kullanılarak klon tespiti yapan [31] yaklaşımlar da mevcuttur.

Ağaç karşılaştırma tekniğinin getirdiği yüksek karmaşıklık ve buna bağlı olarak uzun sürede sonuç üretme sorunununa Koschke ve diğerleri, soyut sözdizimi alt ağaçlarının serileştirilerek AST düğüm sekanslarına dönüştürüldüğü bir yaklaşım önermişlerdir [32]. Dönüşüm sonucu oluşan büyük uzunluktaki AST düğüm sekansı, yalnızca sözdizimsel kapalı sekansların kalması amacıyla sözdizim bölgelerine göre bölümlendirilirler [14].

Kaynak kodun ağaç gösteriminde değişken isimlerinin ve diğer simgelerin soyutlaştırılması ile yüksek düzeyli klon tespiti sağlamaktadır. Tip 1, 2 ve 3 klonları için uygun bir yöntemdir.

2.5.5 Program bağımlılık grafı-tabanlı yaklaşım

Programın metinsel sıralamasının önemli olduğu simge-tabanlı ve sözdizim-tabanlı yaklaşımlarınlarının yanı sıra, kod metninden bağımsız statik program analizi kullanarak programın anlamsal tekrar kullanımını araştıran teknikler mevcuttur.

Kaynak kodun program bağımlılık grafı (Program Dependency Graph - PDG) olarak sergilenmesine dayalı, program bağımlılık ağacı yaklaşımında, düğümler açıklama ve ifadeleri simgeler iken; kenarlar, kontrol ve veri bağımlılıklarını sembolize etmektedir [33]. Graf üzerindeki eşbiçimli alt graflar, klon olarak tanımlanırlar.

Komondoor ve diğerleri tarafından önerilen [34], PDG-DUP isimli PDG tabanlı klon tespit aracında, program dilimleme tekniği kullanılarak eşbiçimli PDG alt grafları saptanırken; Krinke tarafından önerilen [35], Duplix aracında, k-uzunluklu parça eşleştirme temelli iteratif yaklaşım kullanılmaktadır. Program bağımlılık grafı üzerine geliştirilen uygulamalardan biri de, Liu ve diğerleri tarafından geliştirilen GPLAG isimli, telif hakkı ihlallerini PDG üzerinden klon tespit yöntemi ile saptayan araçtır [36].

Tip 1, 2 ve 3 klonlarının tespitinde kullanılmaktadır.

2.5.6 Melez yaklaşımlar

Şimdiye dek bahsedilen karşılaştırma tekniklerine ek olarak, sözdizimsel yaklaşım ile anlamsal yaklaşımı birleştiren [37] ve sözdizimsel yaklaşım ile soyut sözdizim ağacı-tabanlı teknikleri birleştiren [32] yaklaşımlar mevcuttur.

Koschke ve diğerleri tarafından önerilen melez yaklaşımda [32], serileştirilen AST düğümlerinden oluşturulan sekans üzerinde yapılan sözdizimsel bölümleme ile; AST düğümlerinin doğrudan karşılaştırılması yerine, sonek ağacı tabanlı algoritma kullanımı ile AST düğümleri simgesel karşılaştırılmaktadır [2]. Bu yaklaşım sayesinde, klon tespit işlemi lineer zaman ve uzayda gerçekleştirilebilmekte, AST tabanlı tekniğe oranla yüksek performans sağlanabilmektedir.

Leitao tarafından önerilen melez yaklaşımda ise, her biri farklı özellikleri araştıran AST metriklerine dayalı sözdizim teknikleri ve anlamsal teknikler özelleştirilmiş karşılaştırma fonksiyonları birleşimi ile kullanılır [37].

Birleştirici nitelikte olan bu teknikler, klon tipi kapsamını ve klon tespit başarımını artırmaktadır.

2.6 Klon Belirleme Algoritmalarının Karşılaştırılması

Kod klonlarını belirlemek üzere önerilen farklı birçok teknik ve bu tekniklerin arkasında kullanılan birçok algoritma, özelleştirilmiş kullanım amacı ve duyarlık-tamlık değerlerine göre değerlendirilebilirler.

Bailey ve Burd tarafından gerçekleştirilen çalışma [38], teknikleri karşılaştırmayı amaçlayan ilk deneysel çalışmalardandır. Karşılaştırmaya dahil edilen çalışmalar; Kamiya [24], Baxter [1] ve Merlo [26] tarafından önerilen klon tespit yöntemleri ile, Jplag [39] ve Moss [40] telif hakkı ihlali tespiti amaçlı araçlardır. Karşılaştırma sonucunda; Baxter tarafından önerilen sözdizim tabanlı CloneDr aracı, %100 ile en yüksek oranda duyarlığı gösterirken, %9 ile en düşük tamlığı üretebilmiştir. Kamiya tarafından üretilen teknik, diğer tekniklere göre en yüksek duyarlık ve tamlık değerlerine sahip olur iken, Merlo tarafından önerilen metrik-tabanlı tekniğin %63 ile en düşük duyarlığı gösterdiği görülmüştür.

Burd ve Bailey tarafından, küçük boyuttaki (16 KLOC) yalnızca bir sistem üzerinde gerçekleştirilen çalışmayı, Bellon ve diğerleri tarafından gerçekleştirilen kapsamlı çalışma takip etmiştir [14].

Bellon ve diğerleri tarafından gerçekleştirilen çalışma, toplam boyutu yaklaşık olarak 850 KLOC olan 4 adet Java, 4 adet C programlama dilinde toplam 8 sistem üzerinde gerçekleştirilmiştir [3]. Burd ve Bailey'e ait çalışmada kullanılan 3 klon tespit aracına ek olarak, simge-tabanlı araç olan Dup, PDG-tabanlı araç olan Duplix ve

metin-tabanlı araç olan Duploc karşılaştırmaya dahil edilmiştir. Burd ve Bailey'e ait çalışmada kullanılan klon tespit araçlarından biri olan CLAN, Bellon'a ait çalışmada metrik çalışmasının yanı sıra, simgesel ve metinsel yönleriyle de ele alınmıştır [3]. Bellon ve diğerlerine ait çalışmaya dahil edilen araçlar ve özellikleri Çizelge 2.1'de görülmektedir.

Çizelge 2.1 : Bellona'a ait Deney Kapsamı.

Katılımcı	Araç	Teknik
Brenda S. Baker	Dup	Simge
Ira D. Baxter	CloneDr	AST
Toshihiro Kamiya	CCFinder	Simge
Jens Krinke	Duplix	PDG
Ettore Merlo	CLAN	Fonksiyon Metrikleri
Matthias Rieger	Duploc	Metin

Çizelge 2.2'de Bellon ve diğerleri tarafından gerçekleştirilen çalışmaya ait sonuçlar [3] özetlenmektedir. Çalışmada karşılaştırmaya dahil edilen araçlar; tespit edilen klon tipi, zaman ve bellek tüketimleri, insanihminler sonucu elde edilen saptamalar ile karşılaştırmalı olarak tamlık ve duyarlık değerleri yönleriyle değerlendirilmişlerdir. Elde edilen sonuçlar, "--" en kötü durumdan başlayarak, "-", "+" ve "++" en iyi duruma dek ölçeklendirilmişlerdir. "?" ise net olmayan sonuçları nitelendirmektedir.

Çizelge 2.2 : Bellon'a ait Deney Sonuçları.

	Baker	Baxter	Kamiya	Krinke	Merlo	Rieger
Klon Tipi	1, 2	1, 2	1, 2, 3	3	1, 2, 3	1, 2, 3
Hız	++	-	+	--	++	?
RAM (bellek)	+	-	+	+	++	?
Tamlık	+	-	+	-	-	+
Duyarlık	-	+	-	-	+	-

Karşılaştırma deneyinde [3], elde edilen sonuçlar incelendiğinde, karşılaştırmayı kazanan bir araç olmadığı değerlendirilmiştir. Simge tabanlı ve metin tabanlı tekniklerin yaklaşık olarak benzer davranış sergiledikleri ve yüksek tamlık değerine sahip oldukları görülmüştür. Merlo ve Baxter'a ait araçların en yüksek duyarlık

oranına sahip olduğu görülürken, Krinke'ye ait aracın yalnızca Tip 3 klonların tespitinde başarılı olduğu gözlenmiştir.

AST tabanlı tekniklerin, yüksek duyarlık değerlerine rağmen, uzun yürütme zamanı harcamaları, simge tabanlı tekniklerde ise sözdizimsel klonların, AST karşılaştırmasına ters şekilde kısa zamanda bulunabildiği sonucu gösterilmiştir.

Tip 1 ve 2 klonlarının, belirsiz durumdaki Tip 3 klonlarına oranla var olan teknikler ile daha doğru ve kolay tespit edilebildikleri, çalışmanın sonucu olarak belirtilmiştir [3].

Rysselberghe ve Demeyer kod yeniden düzenlemeyi farklı açılardan ele alarak, metin-tabanlı, simge-tabanlı ve metrik-tabanlı teknikleri karşılaştırmışlardır [41]. Yeniden düzenleme aracı ile işlenebilirliğe "uygunluk", yeniden düzenlenecek klon adayları arasındaki "öncelik", aracın ürettiği sonuçlara "güvenilirlik" ve projenin tamamına ya da tek bir sınıfa "odaklanma" yönleriyle inceledikleri tekniklere dair sonuçlar Çizelge 2.3'te görülebilmektedir.

Çizelge 2.3 : Rysselberghe ve Demeyer'a ait Deney Sonuçları.

Kriter	En Uygun Teknik
Uygunluk	Metrik-tabanlı
Öncelik	Fark bulunmamaktadır
Güvenilirlik	Metin-tabanlı
Odaklanma	Fark bulunmamaktadır

Brutink ve diğerleri tarafından C dilindeki programlarda bulunan enine kesen ilgilerin saptanmasında, CCFinder (Kamiya) [24], ccdiml (Baxter'a ait tekniğin varyasyonu) [1] ve PDG-DUP (Komondoor) [34] klon tespit araçlarını kullanarak, bu araçları hata işleme, izleme, ön ve son durum kontrolü ve bellek hata işleme yönleriyle karşılaştırmışlardır [42]. Çalışmaları, ccdiml ve CCFinder araçlarının boş işaretçi ve hata işleme ilgilerini bulurken, PDG-DUP aracının izleme ve bellek hata işleme ilgilerini efektif olarak bulabildiğini göstermiştir [42].

3. KULLANILAN TEKNİK VE ARAÇ

Çalışmada kullanılan kod klonu tespit aracı, seçilen araca ait özellikler, seçim kriterleri ve aracın kullandığı teknik detaylı olarak bu bölümde açıklanmaktadır.

3.1 Klon Tespit Aracı Özellikleri

Çalışmada, simge tabanlı kod klon tespit sistemi olan CCFinder klon tespit aracı kullanılmıştır.

CCFinder klon tespit aracı, simge-tabanlı algoritması ve kullandığı birçok optimizasyon tekniği ile geniş ölçekli yazılım sistemleri için verimli ve ölçeklenebilir bir çözüm niteliği taşımaktadır. Kamiya ve diğerleri, yaptıkları durum çalışmasında, toplamda 10 milyon kod satırlık bir yazılım sistemi üzerindeki denemelerinin 650MHz Pentium III işlemci ve 1GB bellek özellikli ortamda 68 dakika sürdüğünü belirtmişlerdir [24]. Çalışmada kullanılacak simülasyon sistemlerinin büyüklükleri göz önüne alındığında, geniş ölçekli sistemler üzerinde hızlı ve doğru çalışan bir araç gerekliliğini, CCFinder bu özelliği ile karşılayarak tercih sebeplerinden birini oluşturmuştur.

CCFinder, Bellon ve diğerlerine ait klon tespit araçları karşılaştırma çalışmasında belirtildiği gibi, diğer araçlara kıyasla, en yüksek tamlık ve doğruluk değerlerine sahiptir [3]. Metin-tabanlı tekniklerin saptayamadığı klonları tespit edebilmesinin yanı sıra, modifikasyonlara karşı gürbüz olması, diğer araçlar ile yapılan karşılaştırmalarda, daha fazla sayıda klon tespit edebilmesi ve daha doğru klonlar tespit edebilmesi yönleriyle de tercih sebebi olmaktadır.

Kamiya ve diğerleri, yaptıkları çalışmada CCFinder klon tespit aracının hem sistemlerin yalnız kendi içlerindeki klonların tespitinde hem de birden çok sistemin birbirleri arasındaki benzerlik ve farklılıkların tespitinde kullanım amacını edindiklerini açıklamaktadırlar [24]. Bu amaç, çalışmada gerçekleştirilmek istenen analiz kapsamını bütünüyle karşılamaktadır. Analiz kapsamı uyumluluğu yönüyle de, CCFinder klon tespit aracı tercih sebeplerinden birini daha karşılamaktadır.

CCFinder, uyumlu olduğu programlama diller bakımından oldukça kapsamlıdır. COBOL, C, C++, C#, Java ve Visual Basic programlama dillerindeki kaynak kod dosyalarını işleyebilmektedir [24]. Ayrıca, diğer dillere de kolayca adapte edilebilen yapısı nedeniyle, çalışmanın ilerleyen aşamalarında analiz edilmek istenen yazılım sistemleri için esneklik sağlaması açısından da tercih edilmiştir.

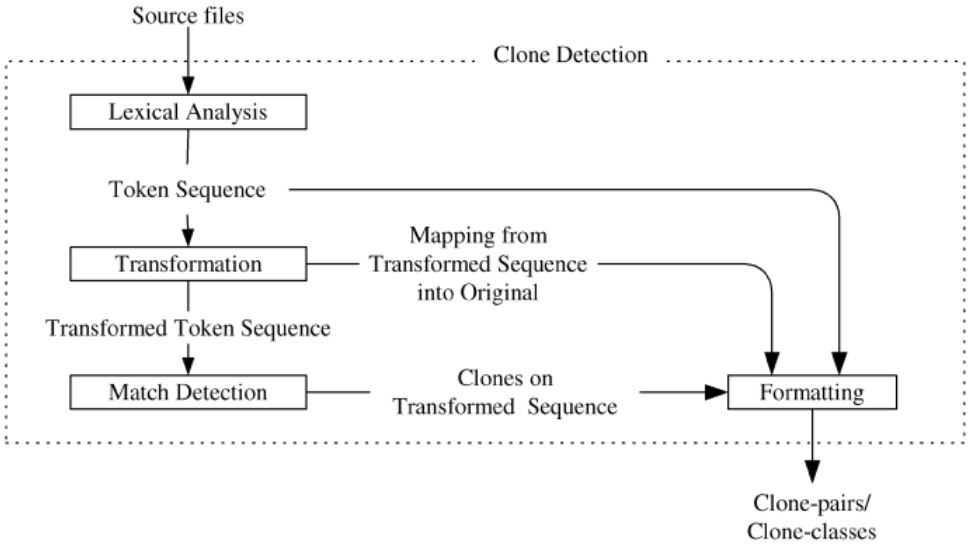
CCFinder, tespit ettiği kod klonu bilgilerinin yanı sıra, sistem karakteristiklerini tanımlamayı kolaylaştıran klon metrikleri de üretmektedir. Klon metrikleri, kod klonlarının bir denklik sınıfı, klonların nerede hangi sıklıklarda görüldüğünün ölçüsü, klonlanmış kodların kaldırılmasıyla ne kadar kod satırı azalacağının ve klon içeren kaynak dosyalarının dosya sistemine ne ölçüde/hiyerarşik uzaklıkta dağıldığının tahmin kaynağı olmaktadır [24]. Sistem bakımı ve yeniden düzenlemesi için önemli olan kod klonlarına işaret eden bu metrikler, çalışmanın amaçladığı tekrar kullanılabilir yazılım bileşenlerinin tespitini kolaylaştıracağı öngörüsü ile klon tespit aracı seçiminde önemli bir referans noktası olmuşlardır.

Araç, en küçük klon uzunluğu, en küçük simge kümesi boyutu, şekillendirici seviyesi, parametre eşleme uygulamaları olmak üzere dört temel parametre kullanmaktadır. En küçük klon uzunluğu, bir kod parçasının klon olabilmesi için gereken en küçük simge sayısını tanımlarken; en küçük simge kümesi boyutu, bir kod klonuna ait kod parçasının simge kümesi boyutunu ifade etmektedir [5]. Blokla ile ayrılmış, bir dış blok ile bölünmemiş, herhangi bir blok içermeyen, herhangi bir sınır içermeyen seviyelerinde olabilecek kod bloğu yapısını tanımlamak üzere kullanılan şekillendirici seviyesi ile değişken ve fonksiyon isimlerindeki farklılığın ihmal edilip edilmeyeceği seçeneğini sunan parametre eşleme seçenekleri sunulmaktadır [5].

Çalışmada, geniş ölçekli yazılım sistemleri analizi için önerilen parametre kümesi (en küçük klon uzunluğu = 100, en küçük simge kümesi boyutu = 10) kullanılmıştır [46]. Parametre değişimlerini göz ardı etmemek ve daha doğru kod klonu adayları görebilmek amacıyla, parametre eşleme seçilmiş olup, olabildiğince düzenli kod blokları elde etmek üzere, bir dış blok ile bölünmemiş şekillendirici seviyesi (yumuşak şekillendirici) seçilmiştir.

3.2 Kod Klonu Tespit Süreci

CCFinder kod klonu tespit aracı tarafından yürütülen simge tabanlı klon tespit süreci, Şekil 3.1'de görüldüğü gibi *Sözcüksel Analiz*, *Dönüşüm*, *Eşleşen Belirleme* ve *Formatlama* olmak üzere dört temel aşamayı içermektedir [24].



Şekil 3.1 : CCFinder simge tabanlı klon tespit işlemi.

3.2.1 Sözcüksel analiz

Sözcüksel Analiz aşamasında, kullanılan programlama dilinin sözcüksel kurallarına uygun olarak, her bir satır simgelere bölünür. Her bir satır için oluşturulan simge sekansları, tek bir simge sekansında birleştirilir. Birleştirme işlemi, klonların çoklu dosya ve dizin yapılarında da tespit edilmesini sağlamak amacıyla gerçekleştirilmektedir. Simgeler arasındaki kaldırılan boşluk, satır sonu, sekme, yorum karakterleri orijinal dosyaların tekrar oluşturulma aşamasında kullanılmak üzere saklanır.

3.2.2 Dönüşüm

Dönüşüm aşaması, dönüşüm kuralları ile dönüşüm ve parametre yer değişimi ile dönüşüm olmak üzere iki alt işlemden oluşmaktadır. Dönüşüm işlemlerinin yanı sıra, dönüştürülen simge sekanslarına ait eşleme bilgileri, formatlama aşamasında kullanılmak üzere kayıt altına alınır.

Dönüşüm kuralları ile dönüşüm işleminde, namespace davranışını, package isimlerini kaldırma gibi tanımlayıcıları düzenleme süreci gerçekleştirilir. Fonksiyon

ayrımlarının belirlenmesi, sınıf yapılarının ayrılması gibi işlemler ile yapıların tanımlanması işlemi gerçekleştirilir. Şekil 3.2'de, dönüşüm kuralları uygulama aşamasında gerçekleştirilen işlem görülebilmektedir.

```
1| Void print_lines(const set<string>& s) {
2|     int c = 0;
3|     set<string>::const_iterator i
4|         = s.begin();
5|     for (; i != s.end(); ++i) {
6|         cout << c << ", "
7|             << *i << endl;
8|         ++c;
9|     }
10| }
11| Void print_table(const map<string, string>& m) {
12|     int c = 0;
13|     map<string, string>::const_iterator i
14|         = m.begin();
15|     for (; i != m.end(); ++i) {
16|         cout << c << ", "
17|             << i->first << " "
18|             << i->second << endl;
19|         ++c;
20|     }
21| }
```

```
1| void print_lines ( const set & s ) {
2| int c = 0 ;
3| Const_iterator I
4| = s . begin ( ) ;
5| for ( ; i != s . end ( ) ; ++ i ) {
6| cout << c << ", "
7| << * I << endl ;
8| ++ c ;
9| }
10| }
11| void print_table ( const map & m ) {
12| int c = 0 ;
13| Const_iterator I
14| = m . begin ( ) ;
15| for ( ; i != m . end ( ) ; ++ i ) {
16| cout << c << ", "
17| << i -> first << " "
18| << i -> second << endl ;
19| ++ c ;
20| }
21| }
```

Şekil 3.2 : Dönüşüm kurallarının uygulanması.

Parametre yer değişimi ile dönüşüm işleminde, ilgili tip, değişken ve sabitlerin özel bir simge ile yer değiştirilmesi ile farklı değişken isimlerine sahip yapıların da klon olarak belirlenebilmesi sağlanmaktadır. Şekil 3.3'de, parametre yer değişimi uygulanmış kod parçası görülebilmektedir.

```
1| $p $p ($p $p & $p ) {
2| $p $p = $p ;
3| $p $p
4| = $p . $p ( ) ;
5| for ( ; $p != $p . $p ( ) ; ++ $p ) {
6| $p << $p << $p
7| << * $p << $p ;
8| ++ $p ;
9| }
10| }
11| $p $p ($p $p & $p ) {
12| $p $p = $p ;
13| $p $p
14| = $p . $p ( ) ;
15| for ( ; $p != $p . $p ( ) ; ++ $p ) {
16| $p << $p << $p
17| << $p -> $p << $p
18| << $p -> $p << $p ;
19| ++ $p ;
20| }
21| }
```

Şekil 3.3 : Parametre yer değişimi ile dönüşüm işlemi.

3.2.3 Eşleşen belirleme

Eşleşen Belirleme aşamasında, dönüştürülen simge sekansının tüm alt katarlarından eşleşen çiftler klon olarak tespit edilirler. Her bir klon çifti, SolBaşlangıç

(LeftBegin), SolBitiş (LeftEnd), SağBaşlangıç (RightBegin) ve SağBitiş (RightEnd) dörtlü grubu ile ifade edilmektedir. Bir klon çifti için; SolBaşlangıç ve SolBitiş, önde gelen klonun başlangıç ve bitiş poziyonlarını ifade ederken, SağBaşlangıç ve SağBitiş onu takip eden diğer klonun başlangıç ve bitiş pozisyonlarını belirtmektedir.

3.2.4 Formatlama

Formatlama aşamasında ise, klon çiftlerinin yerleri, orijinal kaynak dosyadaki satır numarasına dönüştürülerek yerleştirme işlemi gerçekleştirilmektedir.

4. ÇALIŞMA ALANI: SİMÜLASYON SİSTEMLERİ

Simülasyon sistemleri, endüstriyel uygulamalarda, tecrübe etmenin zor, olanaksız ya da yüksek maliyetli olduğu gerçek dünya koşullarını incelemek amacıyla çok sayıda uygulama alanında yaygın olarak kullanılması yönüyle yüksek öneme sahiptirler. TÜBİTAK bu önemi, uzun yıllar boyunca farklı uygulama alanlarında (tren, denizaltı, hava trafik kontrolü gibi…) deneyimlemiştir.

Simüle edilen sistemlerin karmaşıklığı nedeniyle simülasyon yazılımlarının boyut ve karmaşıklığı artmaktadır. Yazılım kalitesini artırmak, geliştirme zaman ve maliyetini düşürmek amacıyla mimari seviyede sistematik yazılım tekrar kullanımı zorunlu hale gelmiştir. Bununla birlikte, artan karmaşıklık ve proje sayısı, farklı sistemler arasında ortak/farklı bileşenler analizini manuel yapmayı yüksek maliyetli hale getirmektedir. Çalışmada, bu tür bir alan analizini sağlamak amacıyla klon tespit yönteminin etkinliği araştırılmıştır.

Dört farklı simülasyon sistemi üzerinde çalışılmıştır. TÜBİTAK tarafından uygulanan gizlilik protokolleri nedeniyle, projelere ait detaylı bilgi sağlanamamaktadır. Çalışma kapsamında analiz edilen sistemler; Proje X, Proje Y, Proje Z ve Proje T olarak isimlendirilmişlerdir. Aşağıdaki çizelgede, incelenen sistemlerin boyutlarına ait bilgiler, sınıf, dosya ve kod satır sayısı olarak görülmektedir.

Çizelge 4.1 : İncelenen Simülasyon Sistemlerine ait Büyüklük Bilgileri.

	Simülasyon Sistemleri			
	Proje X	Proje Y	Proje Z	Proje T
Sınıflar	1,440	1,317	10,877	2,012
Dosyalar	992	2,085	14,590	3,289
Satırlar	192,073	356,404	3,213,352	505,074

Proje Z ve Proje T, Yüksek Düzeyli Mimari (HLA - High Level Architecture) temelinde aynı mimari stili kullanmaktadırlar [43]. Yüksek Düzeyli Mimari,

simülasyon sistemlerinin birlikte çalışarak, daha büyük bir simülasyon sistemi oluşturulmasını sağlayan bir mimari sistem [44] olup, günümüz simülasyon sistemlerinde yaygın olarak kullanılmaktadır.

Proje T'ye ait geliştirme ekibi, Proje Z'ye ait geliştirme ekibinin bir alt grubu durumundadır. Çalışma alanı olarak da birbirine benzer olan bu iki simülasyon projesinden, Proje Z, geliştirme zamanı olarak diğer projeden önce durumdadır.

Proje X, Yüksek Düzeyli Mimari kurallarını uygulamayan, ancak bu mimarinin bazı özelliklerinden yararlanan, diğer projelerden bütünüyle farklı bir geliştirme ekibine sahiptir. Proje Y, bağımsız ve ayrı bir geliştirme ekibine sahip iken, Yüksek Düzeyli Mimari yapıyı işletmeyen özel bir uygulama alanına ait simülasyon motoru altyapısına sahiptir.

Çalışmada, özellikleri belirtilen dört simülasyon projesi, ortak bileşenleri tanımlamak amacıyla CCFinder kod klonu tespit aracı kullanılarak incelenmiştir.

5. ANALİZ SÜRECİ VE DENEYSEL SONUÇLAR

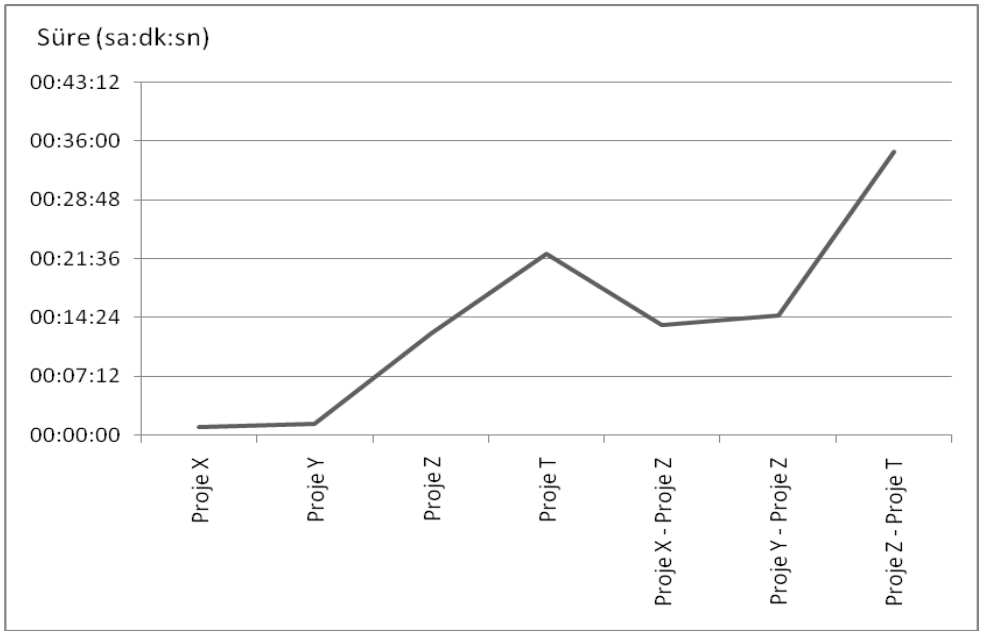
Çalışmanın yürütüldüğü analiz ortamı, yürütülen analiz süreci ve bu süreçten elde edilen deneysel sonuçlar, bu bölümde açıklanmaktadır.

Analiz süreci iki aşamada gerçekleştirilmiştir. İlk adımda, her bir sistemin kendi içindeki kod klonlarının dağılımı incelenmiştir. Ardından, farklı sistemler arasındaki kod klonlarının analizi yapılmıştır.

5.1 Analiz Ortamı

Analiz işlemi için Microsoft Windows XP Professional işletim sistemi yüklü, Intel Core 2 Duo CPU, 1.18 GHz, 2.96 GB RAM özellikli DELL INSPIRON N5030 dizüstü bilgisayar kullanılmıştır.

Özellikleri verilen ortamda gerçekleştirilen analizlere dair süre bilgileri Şekil 5.1'de görülebilmektedir.



Şekil 5.1 : Analiz Süreleri.

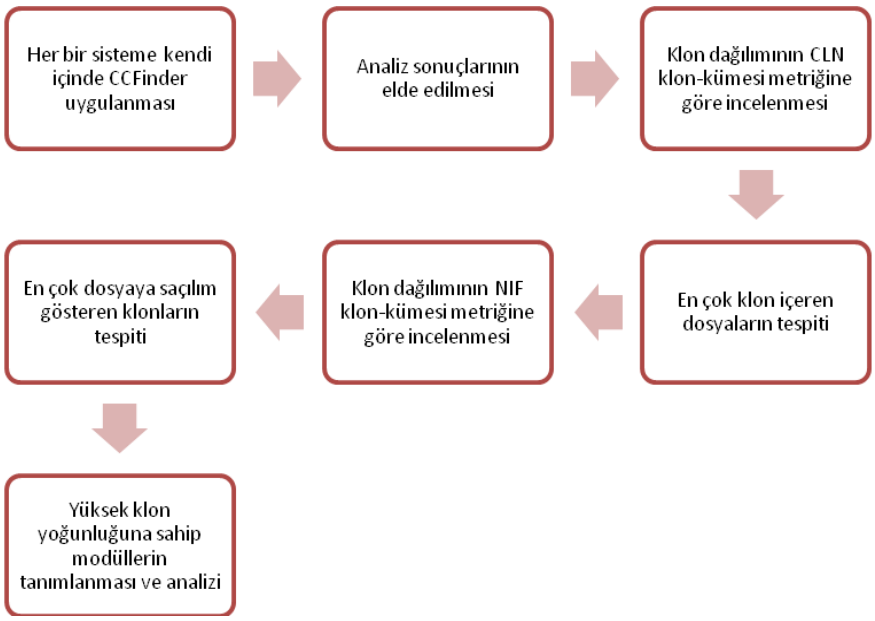
Genel olarak, analiz edilen proje büyüklüğü arttıkça, analiz süresinin de buna bağlı olarak arttığı görülmüştür. İstisnai bir durum olarak; Proje T'nin büyüklüğü Proje Z'den küçük olmasına rağmen, işlem süresinin Proje T'den uzun olduğu ölçülmüştür. Bu durum, Proje T'de bulunan kod klonu büyüklüklerinin diğer projelere oranla daha büyük olması nedeniyle, zaman alan işlem süresi gerektirdiği değerlendirilmiştir.

5.2 Sistemlerin Kendi İçlerindeki Klonlar

Sistemlerin kendi içlerindeki kod klonlarını incelemek amacıyla, aracın sunduğu metrik kümesine ait iki adet metrik kullanılmıştır [24]:

- *CLN*: Kaynak kod dosyasındaki klon kümelerinin sayısını belirten metriktir. Bu metrik, her bir kaynak dosyadaki klon birikimini göstermektedir.
- *NIF*: Bir ya da daha fazla sayıda kod klonu parçası içeren kaynak dosya sayısını belirten metriktir. Bu metrik, her bir kod klonunun kaynak dosyalar arasındaki dağılım miktarını ölçmektedir.

Sistemlerin kendi içlerindeki klonların tespiti için uygulanan analiz süreci akışı, Şekil 5.2'de görülebilmektedir:

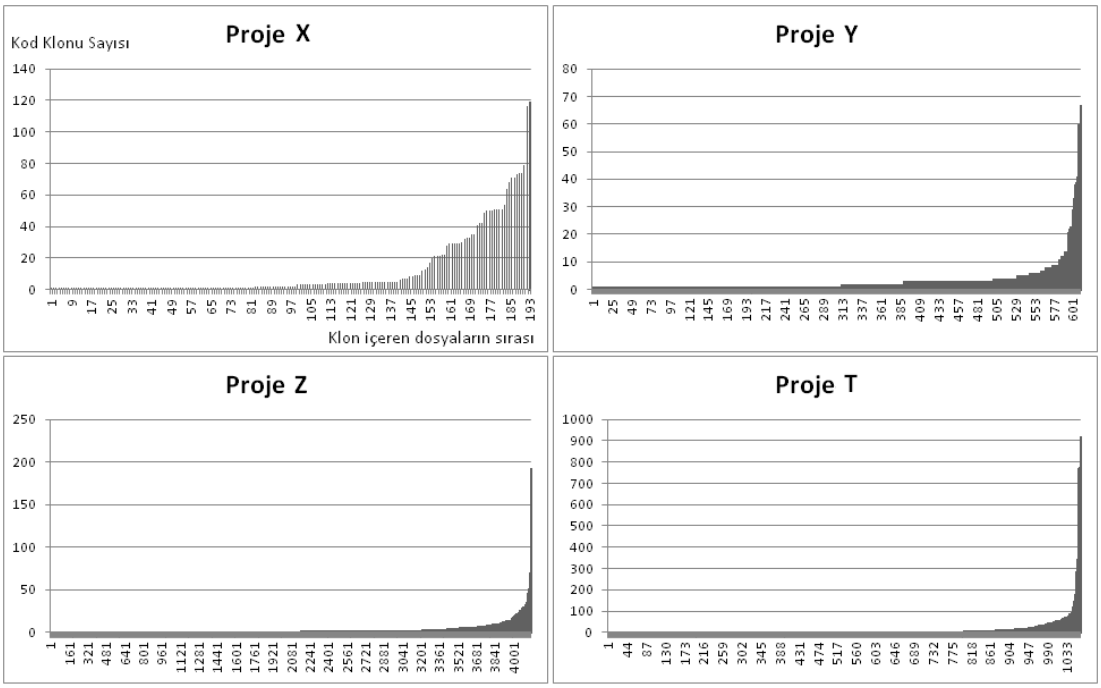


Şekil 5.2 : Sistemlerin kendi içlerinde gerçekleştirilen analiz süreci.

Her bir sisteme ayrı ayrı uygulanan CCFinder aracından elde edilen sonuçlara göre, CLN metriği temel alınarak, kod klonlarının yoğunluğu incelendi. Tespit edilen kod klonlarının çoğunu barındıran dosyalar tanımlandı. Ardından, NIF klon kümesi

metriği temel alınarak, kod klonlarının dağılımı incelendi. Kaynak dosyaların çoğuna saçılan kod klonları tanımlandı. Son aşamada, proje içinde tekrar kullanılabilir bileşen adaylarını saptamak amacıyla yüksek klon yoğunluğuna sahip yazılım modülleri manuel analiz edildi ve elde edilen analiz sonuçlarına göre tekrar kullanılabilir bileşen adayları tanımlandı.

Şekil 5.3'de, her bir proje için, kod klonlarının sistemin kaynak dosyalarına dağılımı incelenebilmektedir.



Şekil 5.3 : Sistemlerin kendi içlerinde klon dağılımları (CLN).

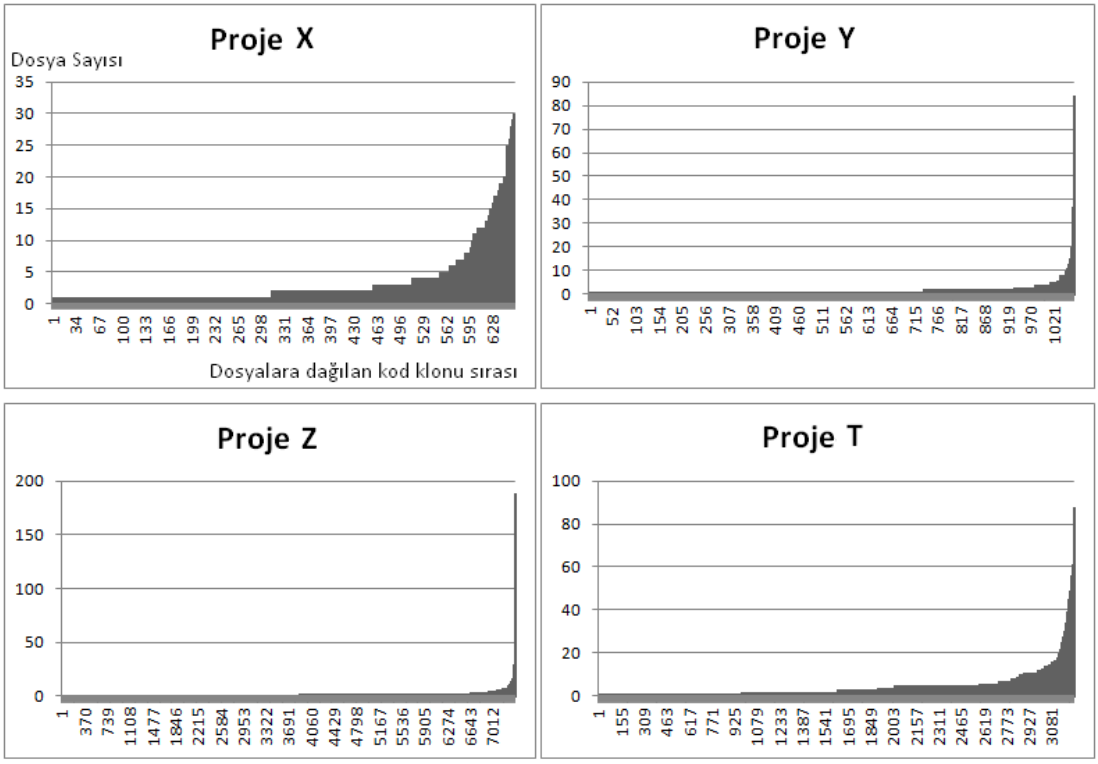
Şekil 5.3'de görüldüğü üzere, kod klonları projelerin tüm dosyalarına eşit olarak dağılmamış, bazı modüllerde büyük oranda birikme göstermişlerdir. Klon dağılımının tüm projeler için Pareto Prensibi [45] ile gerçekleştiği görülebilir.

Kod klonlarının büyük çoğunluğunu içeren yazılım modüllerinin, Kullanıcı Grafik Arayüzü ile ilişkili olduğu gözlenmiştir. Gerçekleştirilen manuel incelemede, bu sonuçların, kullanıcı grafik arayüzü bileşenleri üzerinde gerçekleştirilen ekleme/güncelleme/silme gibi tekrar eden kullanıcı grafik arayüzü işlemleri ile bu bileşenlere mesaj gönderim yapıları nedeniyle oluştuğu görülmüştür.

Projelerin kendi içlerindeki kod klonlarının, proje içi fonksiyonel tekrar kullanımlardan da kaynaklandığı görülmüştür. Bu tekrar kullanımlar, aynı kaynak dosya içinde bulunabildiği gibi, projenin farklı kaynak dosyaları içinde de

bulunabilmektedir. Dosya/Veri tabanı okuma/yazma işlemleri, kayıt günlüğü oluşturma işlemleri, bu tür tekrar kullanımlara örnek verilebilir. Ayrıca, iyi kullanılmamış ya da kullanılmamış kalıtım özelliğinden kaynaklanan birbirine benzer sınıf tanımlamaları mevcut olduğu görülmüştür. Benzer geliştirilmiş sınıflar yapılarının da klon oluşumuna neden olduğu saptanmıştır.

Şekil 5.4'de, NIF klon-kümesi metriğine göre, kod klonlarının dosyalara dağılımları görülebilmektedir.

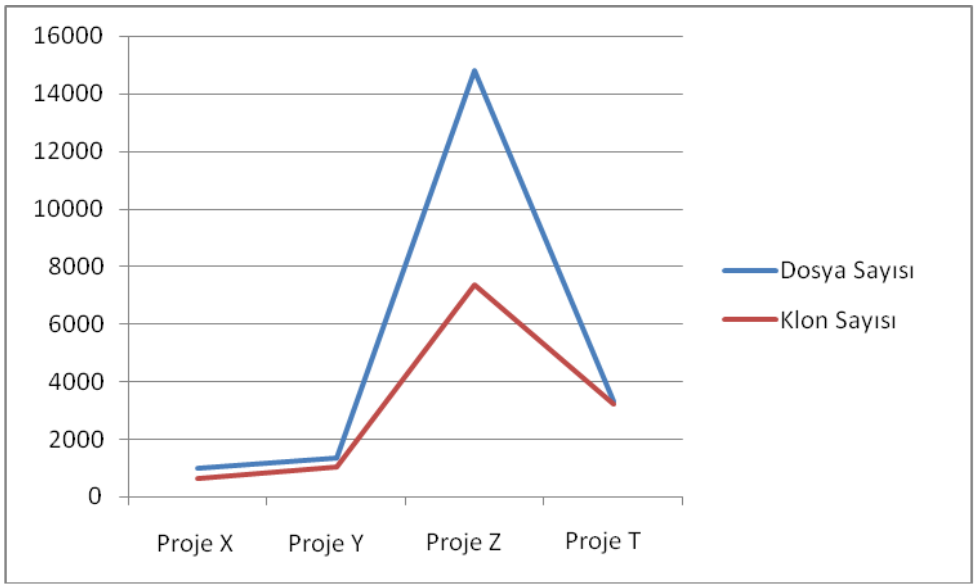


Şekil 5.4 : Sistemlerin kendi içlerinde klonların dosyalara saçılımı (NIF).

Şekil 5.4'e göre, Proje X'de, en çok dosyada bulunan kod klonlarının 628'den büyük sıra numarasına sahip olduğu görülmektedir. Bu kod klonlarının, metot çağrılarına dinamik yönlendirme için kullanılan *if* bloklarının tekrar ettiği bölgelere saçıldığı görülmüştür. Proje Y'de saçılım gösteren kod klonlarının, simülasyon motoru altyapısına gönderilen ilklendirme mesajları ile ilişkili olduğu gözlenmiştir. Diğer projelerde, ilklendirme işlemleri mimari bir katmanda modülarize edilmiştir. Proje Y'de bu işlem için herhangi bir mimari katman bulunmaması, ilklendirme mesajlarının dosyalara saçılımı ile sonuçlanmıştır. Proje Z ve Proje T'de en çok saçılım gösteren kod klonları, modüllerin etkileşim yaratımı ve federasyona kayıt olma bölümleri olduğu görülmüştür. Yüksek düzeyli mimari ile uyumlu olarak;

federeler, simülasyon öğeleri arasındaki karşılıklı olay değişimi için gerçek zamanlı altyapı ile etkileşmelilerdir [43]. Bu sebeple, modüllerdeki bu etkileşimlerin yaratım ve kayıt bölümleri, en çok saçılan klonlara neden olmuşlardır. Genel olarak, kullanıcı grafik arayüzü bileşenleri gerçekleştirim yapısının, konu edilen tüm sistemlerde saçılan klonlara sebep olduğu gözlenmiştir.

Klon birikim ve saçılım analizlerinin yanı sıra, analiz edilen sistemlerin büyüklükleri ile bu sistemlerdeki klonlama oranları incelenmiştir. Şekil 5.5'te, klon sayısının dosya sayısı ile ilişkisel dağılımına ait grafik görülmektedir.



Şekil 5.5 : Proje Büyüklüğü – Klon Sayısı İlişkisi.

Proje büyüklüğünün, klonlama oranını tahmin etmede önemli bir faktör olduğu görülebilmektedir. Şekil 5.5'te görüldüğü üzere, sistem büyüklüğü arttıkça, sistem içindeki kod klonu sayısı da artmaktadır. En büyük kod satırına sahip olan Proje Z içinde aralıklarla yapılan kod yeniden düzenleme çalışmaları sayesinde, diğer projelerdeki dosya sayısı – klon sayısı paralel gelişimi, bu projede görülmemiştir. Kod yeniden düzenleme çalışmaları sayesinde, Klon sayısı/Dosya sayısı oranı, diğer projelere göre düşüktür.

5.3 Sistemler Arası Klonlar

Sistemler arası klonların tespiti için bir dizin hiyerarşisi içinde bulunan kod klonunun kaynak kod parçalarının uzaklık değerini veren *RAD* klon-kümesi metriği kullanılmıştır [46]. Tek bir kaynak dosyada yer alan kod klonları için *RAD* değeri 0

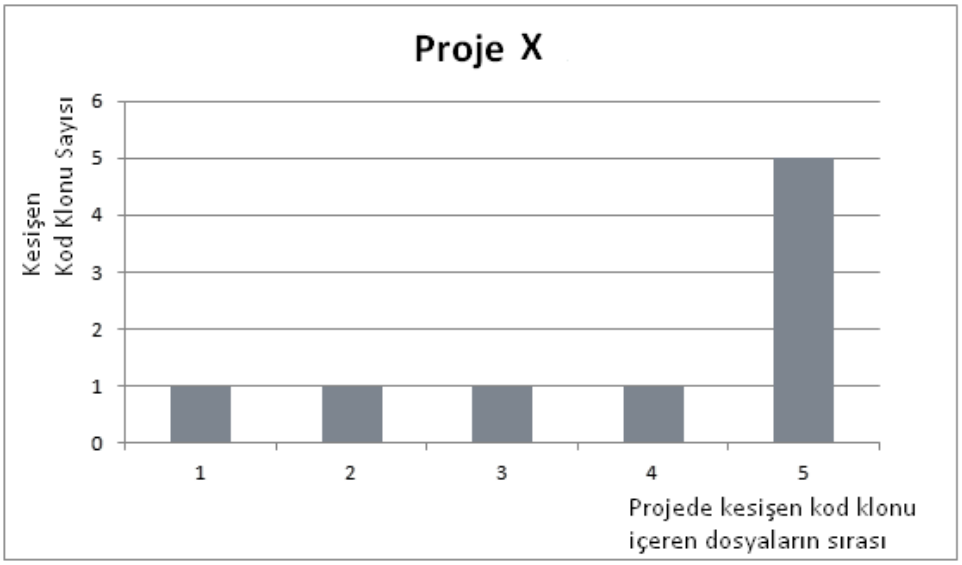
iken, aynı dizin içinde bulunan kod klonları için bu değer 1 olmaktadır. *RAD* değerinin 1'den büyük olması, kod klonuna ait kod parçalarının dizin hiyerarşisi içinde saçıldığını göstermektedir.

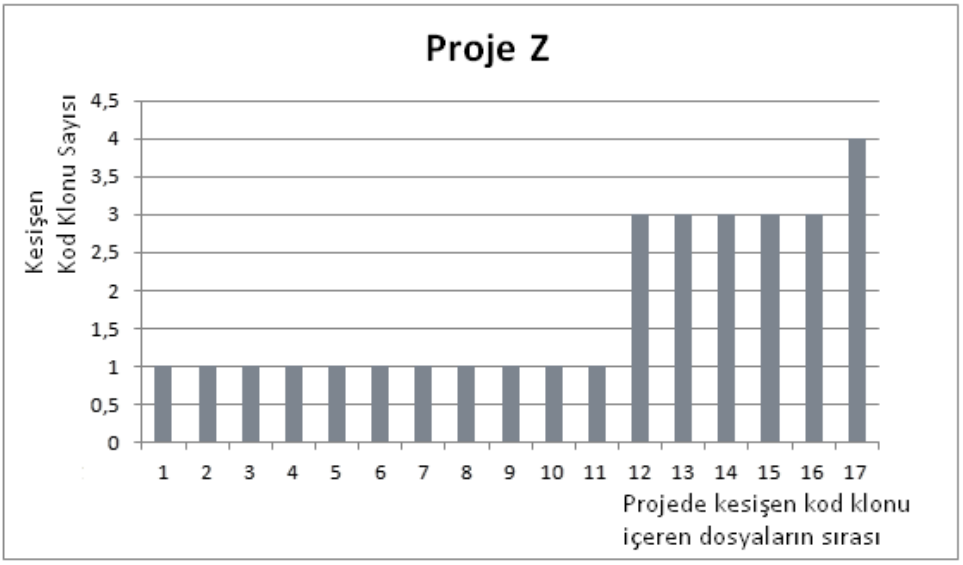
Dört simülasyon sistemi ikili olarak Şekil 5.6'da görülen akış ile analiz edilmiştir:

Şekil 5.6 : Sistemler arasında ikili gerçekleştirilen analiz süreci.

Şekil 5.6'da görüldüğü gibi, analiz süreci kaynak kod dosyalarının her bir sistem için ayrı ayrı tek bir dizinde birleştirilmesi işlemi ile başlar. Bu adım, *RAD* klon metriğinden yararlanabilmek amacıyla, kendi içlerinde projeye özgü çoklu dizin yapısına sahip olan her bir projenin kendine ait tek bir dizinde birleştirilmesi ile farklı dizinde olduğu görülen kod klonunun farklı projede de olduğunu çıkarsamak için gerçekleştirilmiştir. Bu işlem, yazılan bir batch betiği ile gerçekleştirilmiştir. Sistemlerin her bir çifti için gerçekleştirilen kod klon tespit işlemi, ayrı dizinlerde yer alan sistemlerin ikili olarak CCFinder ile analiz edilmesi ile devam etmektedir. *RAD* metrik değerine göre, birden büyük olan kod klonları ayrıştırılarak iki sistem arasında kesişen kod klonları tespit edilmiştir. Kesişen klonların modül-tabanlı dağılımı manuel analiz edilerek, analiz edilen iki sistem arasındaki tekrar kullanılabilir bileşen adayları tanımlanmıştır.

Şekil 5.7, Proje X ve Proje Y arasında kesişen klonların, her bir proje içindeki dağılımını göstermektedir.

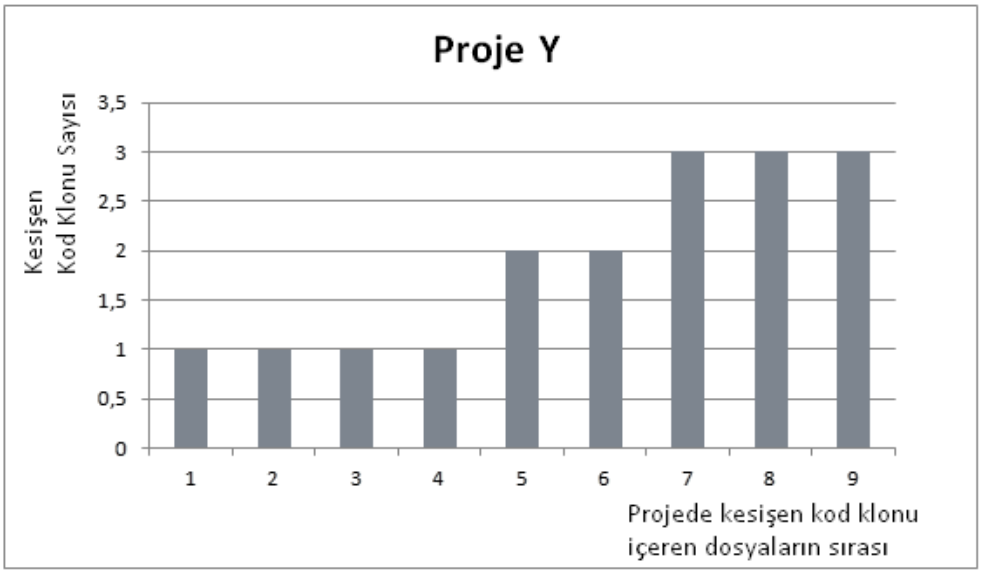


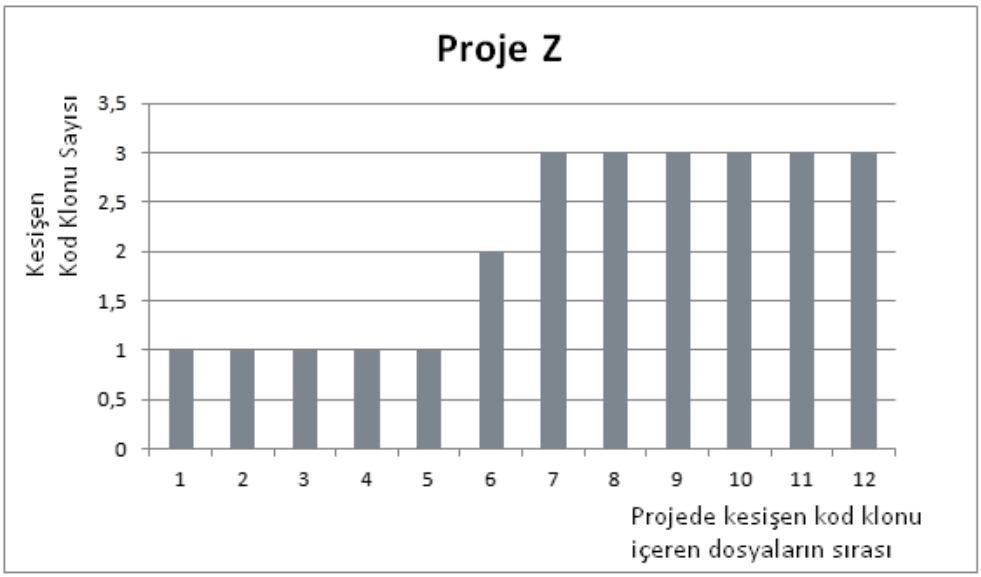


Şekil 5.7 : Kesişen klonların (X-Z) her bir sistemdeki dağılımı.

Kesişen kod klonlarının çoğunlukla, uygulama alanına özgü bileşen davranışlarının ayarlandığı modüllerde yer aldığı görülmüştür. Şekil 5.7'de görüldüğü gibi, kesişen klonların bir kümesi, Proje X'de belirli bir modülde toplanmıştır. Analizin bu aşamasında, uygulama alanına özgü algoritma gerçekleştiriminin her iki projede de var olduğu görülmüştür. Bu algoritmanın, her iki projede de kullanılan bir alan bileşeninin dinamik model kısmına ait olduğu tespit edilmiştir.

Proje Y ve Proje Z arasında kesişen klonlara ait analiz sonuç grafiği Şekil 5.8'de görülebilmektedir.

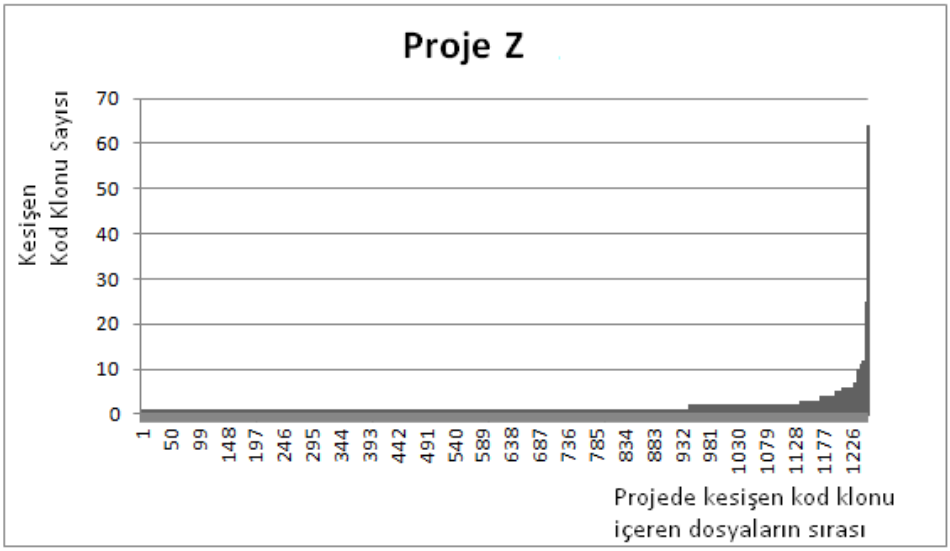


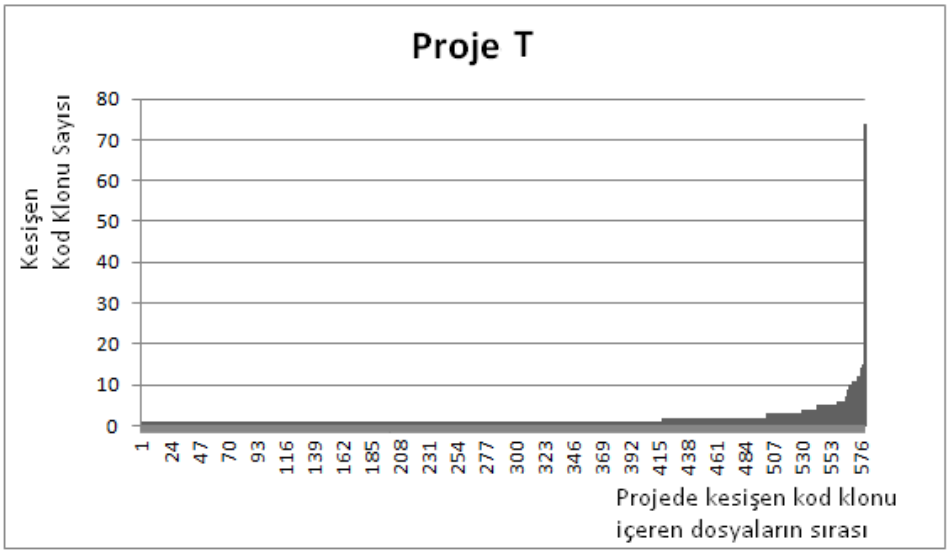


Şekil 5.8 : Kesişen klonların (Y-Z) her bir sistemdeki dağılımı.

Kesişen klonların yine, uygulama alanına özgü bileşen davranışlarının ayarlanması ile ilgili modüllerde olduğu görülse de, bu proje çifti için büyük bir ortaklık ve tekrar kullanılabilir bileşen tanımlanamamıştır. Bu durumun, ilgili iki projenin bütünüyle farklı uygulama alanlarına, simülasyon mimarilerine/altyapılarına ve geliştirme takımlarına sahip olması nedeniyle gerçekleştiği değerlendirilmiştir. Sözü edilen farklılıkların, kesişen herhangi bir klon kümesine rastlanmamasına neden olduğu çıkarsanabilmektedir.

Proje Z ve Proje T ile gerçekleştirilen analiz sonucu tespit edilen kesişen klonların projeler içindeki dağılımına ait grafikler, Şekil 5.9'da görülebilmektedir.





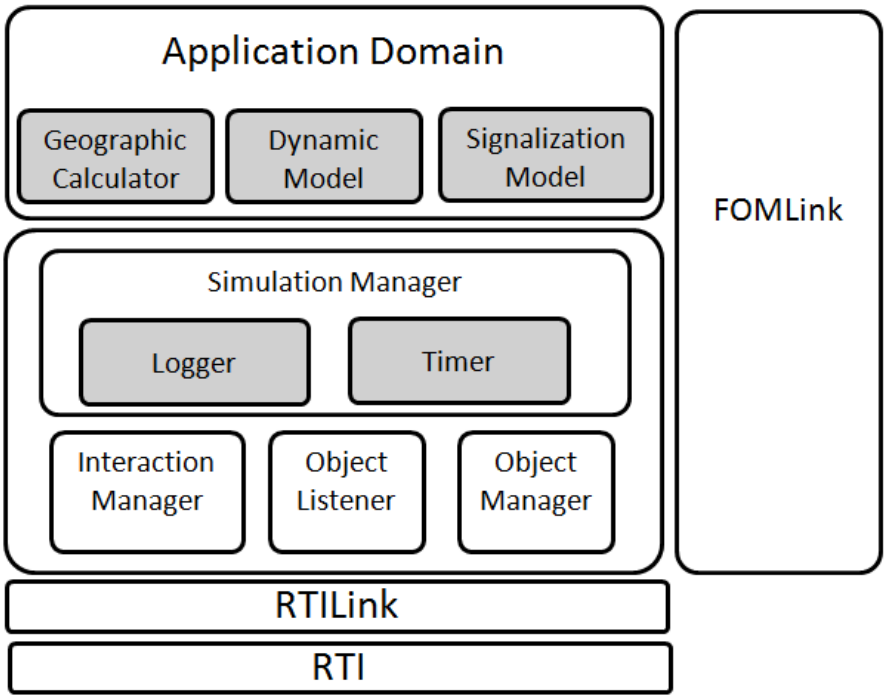
Şekil 5.9 : Kesişen klonların (Z-T) her bir sistemdeki dağılımı.

En yüksek klonlama oranı, bu proje çifti için bulunmuştur. Proje Z ve Proje T, Yüksek Düzeyli Mimari kullanıp, benzer uygulama alanları için geliştirilmişlerdir. Ayrıca, söz konusu iki projenin geliştirme takımları kısmen benzerdir. Bu yönleriyle, ilgili iki projenin en yüksek kesişen klon oranına sahip olması normal bir sonuç olarak değerlendirilebilir. Bununla birlikte, kesişen klonların yukarıdaki grafikte de görüldüğü üzere, küçük bir modül grubunda toplandığı görülmüştür. Bu modüllerin, uygulama alanı bileşenlerinin sinyalizasyon model fonksiyonları ile simüle edilen sistemin dinamik modeline özgü algoritmaları gerçekleştirmekle yükümlü oldukları gözlenmiştir. Bu modüllerden birkaçı, yüksek düzeyli mimarinin bir parçası olan federasyon nesne modeli (Federation Object Model - FOM) işlemlerinden

sorumludur. Yüksek düzeyli mimaride, tüm veriler, federeler arasında doğru iletişim sağlamak amacıyla nesne model taslağında tanımlanan her bir veri tipine göre kodlanmalı ve çözümlenmelidir [43]. Bu kodlama ve çözümleme kurallarının gerçekleştirimi, kesişen klon olarak tespit edilen klonlara neden olmuşlardır.

5.4 Referans Mimari Tanımlama/Detaylandırma

Yüksek düzeyli mimari temelli projeler olan Proje Z ve Proje T ile yapılan analize göre, birçok tekrar kullanılabilir bileşen tanımlandı. Önceden tanımlanmış referans mimari [44] üzerinde yapılan çalışma ile, bu iki proje için elde edilen analiz sonuçlarına uygun olarak detaylandırma çalışması gerçekleştirilmiştir. Yeni eklenen gri gölgeli bileşenler ile detaylandırılan referans mimari, aşağıdaki Şekil 5.10'da görülebilmektedir.



Şekil 5.10 : Tanımlanan tekrar kullanılabilir bileşenler ile referans mimari.

Yüksek düzeyli mimaride [43], FOM, federasyondaki federeler arasında karşılıklı değiştirilen bilginin özel bir tanımlamasıdır. FOMLink, yüksek düzeyli mimaride tanımlanan şekliyle gerçek nesne sınıflarını üretmekten sorumludur. Gerçek zamanlı altyapı (RunTime Infrastructure - RTI), federasyon koşumu sırasında senkronizasyon ve bilgi değişimi için paylaşılan bir arayüz hizmeti sağlamaktadır. Bu mimaride [44], RTILink, sistemi gerçek zamanlı altyapıya bağlayan bir soyutlama katmanıdır.

Etkileşim Yöneticisi (Interaction Manager), uygulama alanı ile RTILink arasındaki etkileşimleri yönetmek ve iletmek ile sorumludur. Nesne Yöneticisi (Object Manager), simülasyon ortamına yayınlanacak olan nesnelerin yaratılma/güncellenme/silinmesinden sorumludur. Nesne Dinleyici (Object Listener) ise bu değişiklikleri kaydetmekten sorumludur.

Kesişen klonların analizi, simülasyon sistemlerinde rutin olarak tekrar kullanılagelen çeşitli fonksiyonelliklerin olduğunu göstermiştir. Bunlar, Simülasyon Yöneticisi (Simulation Manager) katmanında, zamanlama ve kayıt alma mekanizmaları iken; Uygulama Alanı (Application Domain) katmanında, uygulama alanı platformlarına dair dinamik ve sinyalizasyon model algoritmaları ile coğrafi hesaplayıcıdır. Bu sonuçlara göre, tanımlı referans mimari, tanımlanan tekrar kullanılabilir bileşenler gri gösterilecek şekilde Şekil 5.10'da belirtildiği gibi detaylandırılmıştır.

Detaylandırmaların geçerliliği ve önemi, alan uzmanları ve ilgili sistemlerde çalışan yazılım mimarları tarafından doğrulanmıştır.

Detaylandırma öncesi mimariye ait bileşenler olan, Simülasyon Yöneticisi, FOMLink, Nesne Yöneticisi, Nesne Dinleyici ve Etkileşim Yöneticisi de çalışmada tanımlanabilen bileşenlerdir.

.

6. SONUÇLAR VE GELECEK ÇALIŞMA

Klon tespit yönteminin simülasyon sistemleri alan analizi için uygulanabilirliğini araştıran bir çalışma gerçekleştirilmiştir. Bu alanda dört farklı simülasyon projesi incelenmiş, uygulama alanı kavramlarının bir kümesi ile tekrar kullanılabilir bileşenler tanımlanmıştır. Bu doğrultuda, yüksek düzeyli mimariyi temel alan bir referans mimari tanımlanmıştır. Yaklaşımın verimliliğini nicel olarak ölçebilmek için tanımlanmış ve kesinleştirilmiş klon kümesi ya da tekrar kullanılabilir bileşenlerin tam ve kesin bir listesi gibi herhangi bir dayanak bulunmamaktadır. Analiz ile elde edilen sonuçların doğruluğu ve yararlılığı, uygulama alanı uzmanları ve yazılım mimarları tarafından onaylanmıştır. Klon tespit yönteminin, alan analizi ve referans mimari tanımlama/detaylandırma için uygulanabilir olduğu görülmüştür. Ayrıca, tanımlanan tekrar kullanılabilir bileşenler ile, simülasyon projelerinin tamamı için yararlı olabilecek "Yazılım Ürün Hattı" aday bileşenleri belirlendi.

Çalışma sürecinde gözlenen diğer sonuçlar olarak, projedeki kod satır sayısının/dosya sayısının fazla olmasının, aynı mimari altyapıya sahip olmanın, benzer geliştirme ekipleri tarafından geliştirilmenin ve benzer alan hizmeti sunmanın klon sayısını artıran etkenler olduğu görülmüştür.

Projelerin kendi içlerinde gerçekleştirilen analizler ile, yazılım kalitesini artıran bakım işlemi kapsamında yeniden düzenleme çalışmalarının yoğunlaşacağı ve yüksek hata olasılığına sahip alanlara referans oluşturuldu. Ayrıca, projelerin kendi içlerindeki klonların iyi kullanılamamış ya da kullanılması gerektiği halde kullanılmamış kalıtım yapıları nedeniyle de oluştuğu görülmüştür. Bu durum, sınıf yapıları kararlarının verildiği tasarım aşamasına bağlanarak, yazılım kalitesi özelliği olarak değerlendirilen iyi kullanılmış kalıtım özelliğinin kaliteli bir tasarımdan üretilebileceğini destekler niteliktedir.

Çalışmanın proje içi analizlere dair çıktıları, TÜBİTAK BİLGEM bünyesinde bir hizmet oluşmasını sağlamıştır. Projelerin kendi içlerindeki analizler ile elde edilebileceği görülen tekrar kullanılabilir kod bölümleri adayları, kod klonu tespit yönteminin projelerde yeniden düzenleme ve kalitelendirme çalışmaları bünyesinde başvurulan bir teknik olmasını getirmiştir. BTE bünyesindeki kimi projeler için belirli aralıklar ile üretilmeye başlanan "Yazılım Kalite Değerlendirme Raporları"nda yazılım metrikleri ölçümlerinin yanı sıra, yeniden düzenleme sürecine referans olması ve kodun mevcut kalitesi hakkında bilgi sağlaması amaçları ile kod klonu tespit yöntemi ile üretilen sonuçlar da sağlanmaktadır. Çalışmadan, enstitü içi bir yazılım kalite hizmeti olarak yararlanılmaktadır. Analiz yönteminin hizmete dönüştürülmesi sırasında, kullanılan aracın sağlamadığı, gereksinim haline gelen bazı yardımcı bileşenlerin araca entegre edilmesi gerekliliği doğmuştur. Bu bileşenler, kaynak dosya karakter formatlayıcı, aynı isimli dosya tespiti, dizin düzenleyici, sonuç raporlayıcı yazılımlar olarak geliştirilmiş ve analiz sürecine dahil edilerek, iyileştirme sağlanmıştır.

Çalışmada, simge-tabanlı klon tespit yönteminden yararlanıldı. Simge-tabanlı teknik, sözdizimsel eş kopyalar olan Tip 2 klonlarının tespiti için yüksek başarıma sahiptir. Yararlanılan teknik ve aracın performans açısından hızlı sonuçlar üretebilme, analiz işlemi için görece az zaman gerektirmesi avantajından yararlanıldı. Ancak, bu teknik ile Tip 3 ve Tip 4 klonları tespit edilememekte olup, yapısal değişime uğratılan ya da mantıksal seviyede olan kod klonları tespit edilememektedir. Gelecek çalışmada, programın mantıksal benzerliklerini saptama üzerine diğer klon tespit yöntemleri incelenecektir. Mantıksal benzerlik tespitini geniş ölçekli sistemlerde verimli ve performanslı gerçekleştirecek çalışma amaçlanmaktadır. Gerçeklenecek çalışmada, mevcut analiz aracında eksikliği hissedilerek araca entegre edilmek üzere geliştirilen yazılım bileşenleri de göz önünde bulundurularak, detaylı ister analizi yapılacaktır. Ayrıca, tüm tecrübe edilecek teknikler, büyüklük ve alan yönleriyle farklı yazılım sistemleri üzerinde denenecektir.

KAYNAKLAR

[1] **Baxter, I., Yahin, A., Moura, L. ve Anna, M.** (1998). Clone detection using abstract syntax trees. Proceedings of the 14th International Conference on Software Maintenance, sf. 368 – 377.

[2] **Roy, C. K. ve Cordy, J. R.** (2007). A survey on software clone detection research. School of Computing TR 2007-541, Queens University, Cilt 115.

[3] **Bellon, S., Koschke, R., Antoniol, G., Krinke, J. ve Merlo, E.** (2007). Comparison and evaluation of clone detection tools. IEEE Trans. on Software Engineering, Cilt 33, sayı 9, sf. 577–591.

[4] **Rajapakse, D. C. ve Jarzabek, S.** (2005) An investigation of cloning in web applications. Special interest tracks and posters of the 14th international conference on World Wide Web, sf. 924.

[5] **Ma, Y. S. ve Woo, D. K.** (2008). Domain analysis of device drivers using code clone detection method. *ETRI Journal*, Cilt 30, sayı 3, sf. 394–402.

[6] **Baxter, I.** (2002). Using clone detection to manage a product line dale churchett. ICSR7 Workshop, sf. 2–4.

[7] **Antoniol, G., Villano, U., Merlo, E. ve Penta, M.D.** (2002). Analyzing Cloning Evolution in the Linux Kernel. *Information and Software Technology*, 44 (13):755-765.

[8] **Roy, C. K. ve Cordy, J. R.** (2008). An Empirical Study of Function Clones in Open Source Software. IEEE Computer Society, sf. 81–90.

[9] **Schulze, S., Apel, S. ve Christian, K.** (2010). Code clones in feature-oriented software product lines. SIGPLAN Not., Cilt 46, sayı 2, sf. 103112.

[10] **Koschke, R., Frenzel, P., Breu, A. P. J. ve Angstmann, K.** (2009). Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, Cilt 17, sayı 4, sf. 331–366.

[11] **Koschke, R.** (2007). Software Clone Detection State-of-the-Art Survey. Sunum Notları, University of Bremen, Germany.

[12] **Rieger, M., Ducasse, S. ve Lanza, M.** (2004). Insights into System-Wide Code Duplication. Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE'04), sf. 100-109, Delft University of Technology, the Netherlands.

[13] **Roy, C. K.** (2009). Detection and Analysis of Near-Miss Software Clones, School of Computing Doctoral Thesis, Queens University, Canada.

[14] **Koschke, R.** (2007). Survey of research on software clones. Duplication, Redundancy, and Similarity in Software, ser. Dagstuhl Seminar

Proceedings, no. 06301. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[15] **Johnson, J. H.** (1996). Navigating the textual redundancy Web in legacy source. Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'96), sf. 7-16, Toronto, Canada.

[16] **Davey, N., Barson, P., Field, S. ve Frank, R.** (1995). The Development of a Software Clone Detector. *International Journal of Applied Software Technology*, 1(3/4):219–236.

[17] **Burd, E. ve Munro, M.** (1997). Investigating the maintenance implications of the replication of code. Proceedings of the 13th International Conference on Software Maintenance (ICSM'97), Bari, Italy.

[18] **Bruntink, M.** (2004). Aspect Mining using Clone Class Metrics. Proceedings of the 1st Workshop on Aspect Reverse Engineering.

[19] **Bruntink, M., Deursen, A., Engelen, R. ve Tourwe, T.** (2005). On the Use of Clone Detection for Identifying Crosscutting Concern Code.Transactions on Software Engineering, Cilt 31(10):804-818.

[20] **Ducasse, S., Rieger, M. ve Demeyer, S.** (1999). A language independent approach for detecting duplicated code. Proceedings of IEEE International Conference on Software Maintenance (ICSM'99), sf. 109–118.

[21] **Karp, R. M. ve Rabin, M. O.** (1987). Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, Cilt 31, sayı 2, sf. 249–260.

[22] **Johnson, J. H.** (1993). Identifying redundancy in source code using fingerprints. Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Cilt 1, ser. CASCON '93, IBM Press, sf. 171–183.

[23] **Baker, B. S.** (1995). On finding duplication and near-duplication in large software systems. Proceedings of the 2nd Working Conference on Reverse Engineering, sf. 86–95.

[24] **Kamiya, T., Kusumoto, S. ve Inoue, K.** (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. on Software Engineering, Cilt 28, sayı 7, sf. 654–670.

[25] **Li, Z., Lu, S., Myagmar, S. ve Zhou, Y.** (2004). Cp-miner: A tool for finding copy-paste and related bugs in operating system code. Operating System Design and Implementation, sf. 289–302.

[26] **Mayrand, J., Leblanc, C. ve Merlo, E.** (1996). Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. Proceedings of the 12th International Conference on Software Maintenance (ICSM'96), sf. 244-253, Monterey, CA, USA.

[27] **Patenaude, J. F., Merlo, E., Dagenais, M. ve Lague, B.** (1999). Extending software quality assessment techniques to java systems. Proceedings

of the 7th International Work-shop on Program Comprehension (IWPC'99), sf. 4956, Pittsburgh, PA, USA.

[28] **Kontogiannis, K., Galler, M. ve DeMori, R.** (1995). Detecting code similarity using patterns. Working Notes of 3rd Workshop on AI and Software Engineering, 6 sf., Montreal, Canada.

[29] **Calefato, F., Lanubile, F. ve Mallardo, T.** (2004). Function Clone Detection in Web Applications: A Semiautomated Approach. *Journal of Web Engineering*, 3(1): 003–021.

[30] **Di Lucca, G., Penta, M. and Fasolino, A.** (2002). An Approach to Identify Duplicated Web Pages. Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC'02), sf. 481–486, Oxford, England.

[31] **Wahler, V., Seipel, D., Gudenberg, J. ve Fischer, G.** (2004). Clone Detection in Source Code by Frequent Itemset Techniques. Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM'04), sf. 128–135, Chicago, IL, USA.

[32] **Koschke, R., Falke, R. ve Frenzel, P.** (2006). Clone detection using abstract syntax suffix trees. Working Conference on Reverse Engineering, IEEEComputer Society Press.

[33] **Ferrante, J., Ottenstein, K. J. ve Warren, J. D.** (1987). The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9(3):319349.

[34] **Komondoor, R. ve Horwitz, S.** (2001). Using Slicing to Identify Duplication in Source Code. Proceedings of the 8th International Symposium on Static Analysis (SAS'01), Cilt LNCS 2126, sf. 40–56, Paris, France.

[35] **Krinke, J.** (2001). Identifying Similar Code with Program Dependence Graphs. Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), sf. 301–309, Stuttgart, Germany.

[36] **Liu, C., Chen, C., Han, J. ve Yu, P.** (2006). GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06), sf. 872–881, Philadelphia, USA.

[37] **Leitao, A. M.** (2003). Detection of redundant code using r2d2. Workshop Source Code Analysis and Manipulation, IEEE Computer Society Press, sf. 183–192.

[38] **Bailey, J. ve Burd, E.** (2002). Evaluating clone detection tools for use during preventative maintenance. Workshop Source Code Analysis and Manipulation, IEEE Computer Society Press, sf. 36–43.

[39] **Prechelt, L., Malpohl, G. ve Philippsen, M.** (2000). Jplag: Finding plagiarisms among a set of programs. Teknik rapor, University of Karlsruhe, Department of Informatics.

[40] **Schleimer, S., Wilkerson, D.S. ve Aiken, A.** (2003). Winnowing: local algorithms for document fingerprinting. Proceedings of the SIGMOD, sf. 76–85.

[41] **Van Rysselberghe, F. ve Demeyer, S.** (2004). Evaluating clone detection techniques from a refactoring perspective. International Conference on Automated Software Engineering.

[42] **Van Rysselberghe, F. ve Demeyer, S.** (2003). Evaluating clone detection techniques. Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications (ELISA'03), 12 sf., Amsterdam, The Netherlands.

[43] **Simulation Interoperability Standards Committee (SISC)** (2000). IEEE standard for modeling and simulation high level architecture-framework and rules. IEEE Std 15162000, sf. i –22.

[44] **Dikenelli, O., Akdemir, C. ve Timar, Y.** (2011). Dağıtık Simülasyon Sistemlerinde HLA ve Federe Yönetim Katmanı. TÜBİTAK BİLGEM Eylül-Aralık 2011, Cilt 3, sayı 7, sf. 16–23.

[45] **Newman, M. E. J.** (2004). Power laws, pareto distributions and zipf's law. Ekim, Cilt 46, sayı x, sf. 28.

[46] **AIST,** "The archive of ccfinder official site.". Alındığı tarih: 01.04.2012, adres: http://www.ccfinder.net/

ÖZGEÇMİŞ

Ad Soyad: Merve ASTEKİN

Doğum Yeri ve Tarihi: Manisa, 02/01/1988

Adres: TÜBİTAK BİLGEM BTE, P.K. 74 Gebze/KOCAELİ

E-Posta: merve.astekin@tubitak.gov.tr

Lisans: İstanbul Teknik Üniversitesi Bilgisayar Mühendisliği

Mesleki Deneyim ve Ödüller:

TUBITAK-BİLGEM-BTE *Ağustos 2010 – Devam Ediyor*
Yazılım Test ve Kalite Değerlendirme Merkezi (B300)
Gebze
Araştırmacı

TUBITAK-BİLGEM-BTE *Nisan 2010 – Ağustos 2010*
Yazılım Test ve Kalite Değerlendirme Merkezi (B300)
Gebze
Yardımcı Eleman

American International Radio Telekomünikasyon Çözümleri A.Ş. *Haziran 2009 – Temmuz 2009*
Sistem Çözümleri Departmanı
İstanbul
Stajyer

Vestel Dijital Üretim Sanayi A. Ş. *Temmuz 2008 – Ağustos 2008*
Donanım AR-GE Departmanı
Manisa
Stajyer

Haziran 2008 – Temmuz 2008

Vestel Dijital Üretim Sanayi A. Ş.
Yazılım AR-GE Departmanı
Manisa
Stajyer

TEZDEN TÜRETİLEN YAYINLAR/SUNUMLAR

▪ **Astekin M.**, Sözer H., 2012: Utilizing Clone Detection for Domain Analysis of Simulation Systems. *10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture*, August 20-24, 2012 Helsinki, Finland.